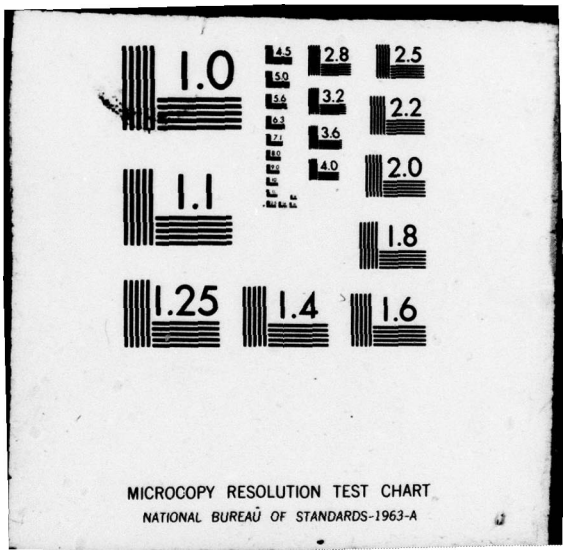
AD-A070 753 HONEYWELL INC MINNEAPOLIS MINN SYSTEMS AND RESEARCH --ETC F/G 9/2
THE GREEN LANGUAGE: AN INFORMAL INTRODUCTION.(U)
APR 79 MDA903-77-C-0331

UNCLASSIFIED NL

1 OF 2
AD
A070753


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

D R A F T

THE GREEN LANGUAGE : AN INFORMAL INTRODUCTION .

LEVEL III

Honeywell, Inc.
Systems and Research Center
2600 Ridgway Parkway, Minneapolis, MN 55413

and

Cii Honeywell Bull
68 Route de Versailles
78430 Louveciennes, France

(11) Apr 79 (15)

(12) 104p.

Contract No. MDA903-77-C-0331

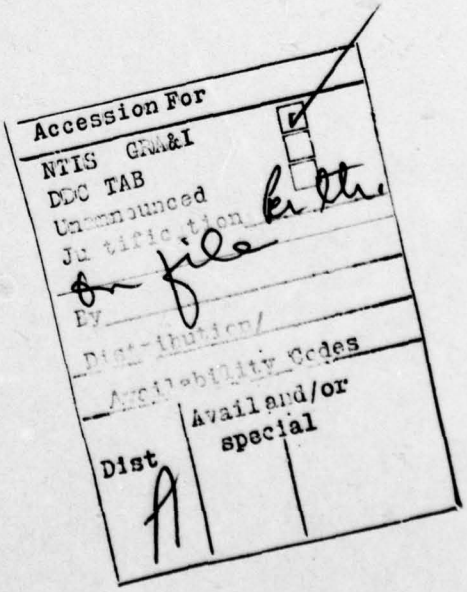
Note: This document is a supplement to the Green Reference Manual. It is presented here in draft form.

402549

The Green Language:

AN INFORMAL INTRODUCTION

TABLE OF CONTENTS



PREFACE

This document is an introduction to the Green programming language. Like any introduction, we emphasize the concepts that form the foundation of the language, rather than attempt to describe the entire language or to provide a self-contained guide for writing programs. Our objective is to convey the essence of the language, mainly through examples.

We assume that the reader has experience in another higher order programming language. No other particular kind of expertise is assumed.

This introduction is organized in three successive levels:

(a) Chapter 1. This opening chapter provides a very brief sketch of the facilities that have an analogue in many other higher order languages. The discussion is based on four small example programs.

(b) Chapters 2 through 4. These chapters provide a much more thorough treatment of the facilities sketched in Chapter 1. In particular, we treat the notion of data types, the basic statements in the language, and the facilities for writing subprograms.

(c) Chapters 5 through 9. These chapters describe the facilities needed in many important applications. In particular, we treat the development of program packages, parallel processing, exception handling, the interface with an implementation, and separate translation.

1. FOUR EXAMPLES

Learning to communicate in a new language is always a challenging experience. At first it may seem as though everything is new and all of it must be understood before there is any hope of using the language at all. After a little experience though, it becomes apparent that there are points of reference to our own language. After a while, we realize that both languages have many common roots.

In introducing you to Green we take advantage of its similarities to languages in common usage, and we urge you as a reader to do the same. In particular, in this chapter we present four small programs. These programs provide a sketch of the features in Green that are common to many other higher order programming languages. In the following three chapters we again sketch these same features, but with much greater detail.

1.1 Temperature Conversion

We write programs in order to interpret and transform data. In writing programs we must describe the nature of the data and give statements for carrying out computations on the data.

Example 1.1 illustrates the structure of a complete program. It has the outline form:

```
restricted(TEXT_IO)
procedure TEMPERATURE_CONVERSION is
   -- a declarative part
   -- describing the data
begin
   -- a statement part
   -- describing the computations
end;
```

Notice how the keywords procedure, is, begin and end form a frame for the program description.

The name of the program is TEMPERATURE_CONVERSION and its form is that of a procedure. A procedure is one form of program unit. The prefix restricted(TEXT_IO) is necessary since the program is using the input output procedures GET and PUT defined in the library package TEXT_IO.

Example 1.1 Converting Temperatures

```
restricted(TEXT_IO)
procedure TEMPERATURE_CONVERSION is
  use TEXT_IO;

   -- A comment.
   -- This program reads a value representing a
   -- fahrenheit temperature and converts it to a
   -- value representing its centigrade equivalent.

   type REAL is digits 7;

   FAHRENHEIT_TEMP, CENTIGRADE_TEMP : REAL;

begin

   GET(FAHRENHEIT_TEMP);
   CENTIGRADE_TEMP := (5.0/9.0) * (FAHRENHEIT_TEMP - 32.0);
   PUT(CENTIGRADE_TEMP);

end;
```

The declarative part of our program includes a type declaration

```
type REAL is digits 7;
```

This declaration specifies that REAL is the name of a type of floating point numbers with 7 digits of precision. Therafter we have the variable declaration

```
FAHRENHEIT_TEMP, CENTERGRADE_TEMP: REAL;
```

This declaration specifies that the two variables used in the program have the properties of real numbers. In general, a declaration associates one or more identifiers with an entity. In this case, the two variable names are objects associated with the type REAL. This is characteristic of all programs: every variable must be declared and associated with a given type.

The statement part of our program begins with a subprogram call

```
GET(FAHRENHEIT_TEMP);
```

This statement calls upon the procedure GET defined in the library package TEXT_IO to obtain a value from some input device and assign the value to the variable FAHRENHEIT_TEMP. The second statement

```
CENTIGRADE_TEMP := (5.0/9.0) * (FAHRENHEIT_TEMP - 32.0);
```

assigns a newly computed value to the variable CENTIGRADE_TEMP. The last statement

```
PUT(CENTIGRADE_TEMP);
```

is like the first and calls upon a subprogram to print the value of the variable CENTIGRADE_TEMP. Notice that each statement and declaration ends with a ";".

Our example also contains four lines of comment. All comments begin with "--" and are terminated by the end of the line. Comments may generally appear anywhere within a program unit.

Procedures such as GET and PUT which are supplied by a package are normally named by dot-notation, for example, TEXT_IO.GET and TEXT_IO.PUT. Here we have inserted a clause use TEXT_IO; at the start of the declarative part and as a consequence GET and PUT can be named directly.

1.2 Counting Change

In many applications we need to store data together. We also need to direct the flow of control in order to perform some computation. These notions are illustrated in Example 1.2. This program reads in six numbers representing the number of pennies, nickels, dimes, etc. given as change, and computes the total value of the change.

The declarative part of our example program introduces four variables of type INTEGER. The predefined type INTEGER is familiar, and one can perform the conventional operations on integers.

Of more interest to us here is the declaration of an array named COIN_VALUE. The type of COIN_VALUE is specified by the type definition

```
array (1..6) of INTEGER;
```

Accordingly, COIN_VALUE denotes a collection of six integer components. The components collectively form an array indexed by the subscripts 1 through 6. The array is initialized with the six integer values

```
(01, 05, 10, 25, 50, 100)
```

These values respectively represent the value in cents of a penny, nickel, dime, quarter, half dollar, and silver dollar.

The fact that COIN_VALUE is specified as constant means that the components of the array must always hold these values, as summarized below.

Array Component	Index	Component Value
COIN_VALUE(1)	1	1
COIN_VALUE(2)	2	5
COIN_VALUE(3)	3	10
COIN_VALUE(4)	4	25
COIN_VALUE(5)	5	50
COIN_VALUE(6)	6	100

Example 1.2 Counting change

```
restricted(TEXT_IO)
procedure COUNT_YOUR_CHANGE is
   use TEXT_IO;

   -- This program reads in 6 integer values, respectively
   -- representing the number of pennies, nickels, dimes, quarters,
   -- half dollars, and silver dollars in coinage.
   -- The program outputs the total value of the coins in dollars
   -- and cents.

   COIN_COUNT,
   TOTAL_CHANGE,
   NUM_CENTS,
   NUM_DOLLARS    : INTEGER;

   COIN_VALUE     : constant array (1..6) of INTEGER :=
                    (01, 05, 10, 25, 50, 100);

begin

   TOTAL_CHANGE := 0;

   for NEXT_COINT in 1..6 loop
      GET(COIN_COUNT);
      TOTAL_CHANGE := TOTAL_CHANGE + COIN_VALUE(NEXT_COIN)*COIN_COUNT;
   end loop;

   if TOTAL_CHANGE = 0 then
      PUT("NO CHANGE");
   else
      NUM_DOLLARS := TOTAL_CHANGE/100;
      NUM_CENTS   := TOTAL_CHANGE mod 100;
      PUT("CHANGE IS $");
      PUT(NUM_DOLLARS);
      PUT(".");
      PUT(NUM_CENTS);
   end if;

end;
```

The statement part of our program is similar to our previous program, except here we have a for loop and an if statement. The for loop has the form

```
for NEXT_COIN in 1..6 loop
   -- statements
end loop;
```

The for loop is controlled by the loop parameter NEXT_COIN. The for loop header also specifies that the loop is to be executed for successive values of NEXT_COIN in the range

```
1..6
```

Thus the statements in the loop are executed six times, and upon each iteration NEXT_COIN is assigned one of the values 1 through 6.

The if statement has the form

```
if TOTAL_CHANGE = 0 then
   -- statements
else
   -- statements
end if;
```

Here we see the choice of one of two groups of statements based on the value of a condition. If the condition testing the value of TOTAL_CHANGE is true, the first group of statements is executed; otherwise the second group is executed.

If we examine our program in a little more detail, we can make a few other points. Consider the statement

```
TOTAL_CHANGE := TOTAL_CHANGE + COIN_VALUE(NEXT_COIN)*COIN_COUNT;
```

Here we see the use of the loop parameter NEXT_COIN, which must have one of the values 1 through 6. Notice that the loop parameter is not declared in the declarative part of the program. A loop parameter is implicitly declared by its appearance in a for loop and has the type INTEGER of the values 1 through 6. Notice also that the operator * is applied before +, since multiplication has a higher precedence than addition.

Finally, we consider the two statements

```
NUM_DOLLARS := TOTAL_CHANGE/100;
NUM_CENTS   := TOTAL_CHANGE mod 100;
```

Here TOTAL_CHANGE/100 gives the integer quotient from the division of TOTAL_CHANGE by 100, and TOTAL_CHANGE mod 100 gives the corresponding integer remainder.

1.3 A Better Way to Count Change

A fundamental aspect in programming is the use of different kinds of objects. Whether the objects are coins, part numbers, tracking positions, or people's names, they must be represented in terms of the constructs of a given programming language.

There are a number of problems with our previous program for counting change. The coins penny through dollar are represented by the integers 1 through 6. Conceptually, coins are simply not numbers. Furthermore, the fact that a dime is represented by 3 even allows us to add a dime to a dime and get 6 (a dollar).

We next revisit our program to count change. Our objective is to illustrate the conceptual unity of declaring a type named COIN, as well as to illustrate the greater security of such types.

The second program for counting change is given in Example 1.3. Its essential differences from the previous example are outlined in

```
procedure COUNT_YOUR_CHANGE is
   ...
   type COIN is (PENNY, NICKEL, DIME, QUARTER, HALF_DOLLAR, DOLLAR);
   ...
   COIN_VALUE : constant array (PENNY .. DOLLAR) of INTEGER :=
                (PENNY   => 01, NICKEL      => 05, DIME   => 10,
                 QUARTER => 25, HALF_DOLLAR => 50, DOLLAR => 100);
   ...
   for NEXT_COIN in PENNY .. DOLLAR loop
      ...
   end loop;
   ...
end;
```

We see here the use of a type named COIN. Unlike the type named INTEGER, the type named COIN is not predefined in the language. It is introduced by the declaration

```
type COIN is (PENNY, NICKEL, DIME, QUARTER, HALF_DOLLAR, DOLLAR);
```

This declaration defines COIN as a type having six values, the identifiers PENNY through DOLLAR.

Example 1.3 Counting Change Using the type COIN

```
restricted(TEXT_IO)
procedure COUNT_YOUR_CHANGE is
   use TEXT_IO;

   -- This program reads in 6 integer values, respectively
   -- representing the number of pennies, nickels, dimes, quarters,
   -- half-dollars, and silver dollars in coinage.
   -- The program outputs the total value of the coins in dollars
   -- and cents.

   type COIN is (PENNY, NICKEL, DIME, QUARTER, HALF_DOLLAR, DOLLAR);

   COIN_COUNT,
   TOTAL_CHANGE,
   NUM_CENTS,
   NUM_DOLLARS : INTEGER;

   COIN_VALUE  : constant array (PENNY .. DOLLAR) of INTEGER :=
                 (PENNY   => 01, NICKEL      => 05,  DIME   => 10,
                  QUARTER => 25, HALF_DOLLAR => 50.. DOLLAR => 100);
begin

   TOTAL_CHANGE := 0;

   for NEXT_COIN in PENNY .. DOLLAR loop
       GET(COIN_COUNT);
       TOTAL_CHANGE := TOTAL_CHANGE + COIN_VALUE(NEXT_COIN)*COIN_COUNT;
   end loop;

   if TOTAL_CHANGE = 0 then
      PUT("NO CHANGE");
   else
      NUM_DOLLARS := TOTAL_CHANGE/100;
      NUM_CENTS   := TOTAL_CHANGE mod 100;
      PUT("CHANGE IS $");
      PUT(NUM_DOLLARS);
      PUT(".");
      PUT(NUM_CENTS);
   end if;

end;
```

The type COIN is used in several places in our program. The first is in the declaration of COIN_VALUE, which has the type definition

```
array (PENNY .. DOLLAR) of INTEGER
```

This definition is analogous to the earlier definition

```
array (1 .. 6) of INTEGER
```

with the important difference that the indices are not denoted by the values 1 through 6 of type INTEGER, but rather by the values PENNY through DOLLAR of type COIN. The components of the array are, of course, integers.

The initialization of the array COIN_VALUE takes advantage of the type COIN in explicitly associating each coin with its corresponding value in cents. In particular, we have

```
(PENNY   => 01, NICKEL      => 05, DIME   => 10,
 QUARTER => 25, HALF_DOLLAR => 50, DOLLAR => 100)
```

This listing of index-value pairs is equivalent to the simple listing of the values

```
(01, 05, 10, 25, 50, 100)
```

but takes the guesswork out of interpreting of the values. Our COIN_VALUE array may thus be summarized as follows:

Array component	Index	Component Value
COIN_VALUE(PENNY)	PENNY	1
COIN_VALUE(NICKEL)	NICKEL	5
COIN_VALUE(DIME)	DIME	10
COIN_VALUE(QUARTER)	QUARTER	25
COIN_VALUE(HALF_DOLLAR)	HALF_DOLLAR	50
COIN_VALUE(DOLLAR)	DOLLAR	100

The next use of COIN is in the for loop

```
for NEXT_COIN in PENNY .. DOLLAR loop
   -- statements
end loop;
```

Here again, the values taken on by NEXT_COIN are not the integers in the range

```
1 .. 6
```

but the enumeration values in the range

```
PENNY .. DOLLAR
```

of COIN. Thus the loop is still executed six times, but on each iteration NEXT_COIN is successively assigned one of the values PENNY through DOLLAR

of type COIN. As before, NEXT_COIN is implicitly declared by its use as the loop parameter, but here NEXT_COIN has the type COIN, the type of PENNY and DOLLAR. As a result, NEXT_COIN can only take on values of type COIN and arithmetic may not be performed on these values.

Each of these uses of the type COIN illustrates a general point. The type COIN represents a conceptual unit introduced by the programmer. Rather than rely on representing coins as integers, the programmer can capture the notion of a coin by giving it a type definition of its own. The use of this type has the added security that once an element has a coin type, it can only take on values of this type.

1.4 Target Practice

The ability to organize programs into subprograms is an important part of every programming language. Furthermore, the ability to parameterize a subprogram allows a programmer to summarize its behavior in terms of its logical inputs and outputs.

These notions are illustrated in the program of Example 1.4. This program reads in the initial X and Y velocities of a projectile, as well as the distance to a target and the height of the target. The program determines whether the projectile will hit the target.

Our interest in this example centers on the subprogram COMPUTE_RISE, which has the following outline:

```
procedure COMPUTE_RISE(V_X, V_Y, DISTANCE: in REAL;  RISE: out REAL) is
   -- local declarations
begin
   -- local statements
end;
```

This subprogram has three input parameters and one output parameter, each of type REAL. Inside the subprogram, the parameters have the names V_X, V_Y, DISTANCE, and RISE. These parameters characterize the behavior of the subprogram. Internally, the subprogram computes a value for RISE based on the three inputs V_X, V_Y, and DISTANCE.

Example 1.4 Target Practice

```
restricted(TEXT_IO)
procedure TARGET_PRACTICE is
   use TEXT_IO;

   -- This program reads in four values, respectively representing
   -- the X and Y velocities of a projectile, the distance to a
   -- target, and the height of the target.
   -- It prints a message indicating whether the projectile will
   -- hit the target or not.

   type REAL is digits 7;


   X_VELOCITY, Y_VELOCITY,
   TARGET_DISTANCE, TARGET_HEIGHT, NET_RISE: REAL;

   G: constant REAL := 32.2

   procedure COMPUTE_RISE(V_X, V_Y, DISTANCE: in REAL;  RISE: out REAL) is
      TIME : REAL;
   begin
      TIME := DISTANCE/V_X;
      RISE := V_Y*TIME - (G/2.0)*(TIME**2);
   end;

begin

   GET(X_VELOCITY);
   GET(Y_VELOCITY);
   GET(TARGET_DISTANCE);
   GET(TARGET_HEIGHT);

   COMPUTE_RISE(X_VELOCITY, Y_VELOCITY, TARGET_DISTANCE, NET_RISE);

   if NET_RISE > 0 and NET_RISE < TARGET_HEIGHT then
      PUT("HIT");
   else
      PUT("MISS");
   end if;

end;
```

This subprogram is used in the main program, which contains the subprogram call

```
COMPUTE_RISE(X_VELOCITY, Y_VELOCITY, TARGET_DISTANCE, NET_RISE);
```

Corresponding to the definition of the subprogram COMPUTE_RISE, the subprogram call contains four arguments, each of type REAL. The first three arguments give the three input values for the procedure. After the call, the fourth argument NET_RISE will take on the output value computed for the parameter RISE.

Our example also illustrates another basic notion. Program units (in this case procedure subprograms) can be nested. In our case we have the structure

```
procedure TARGET_PRACTICE is
   ...
   procedure COMPUTE_RISE (parameters) is
      ...
   end;
   ...
end;
```

Notice that the inner program unit COMPUTE_RISE refers to the gravitational constant G that is declared in the outer unit. Notice also that the inner unit contains the declaration of a local variable TIME. Thus we see the dual role of nesting. An inner unit may refer to global information given in an outer unit, and it may also contain information that is (and should be) entirely internal.

2. DESCRIBING DATA

In every application we have many kinds of objects, each with different properties. To reflect an application properly, we must describe real world objects and their properties in terms of the constructs of a given programming language. Moreover, we must not only choose an appropriate description for an entity, but must make sure that an operation validly performed in a program has a meaning in terms of the real world objects and operations.

For instance, we can perform the conventional arithmetic operation on numeric data, but while we can describe dates as integers and can subtract two dates to get an interval of time, it does not make any sense to take the square root of a date or to multiply a date by a velocity.

We thus see two critical programming issues:

- The need to describe objects and their properties with precision and clarity.
- The need to guarantee that the operations over objects do not violate their intrinsic properties.

This leads us to the notion of types.

2.1 The Notion of Types

We begin with two examples:

```
type DAY  is (MON, TUE, WED, THU, FRI, SAT, SUN);
type COIN is (PENNY, NICKEL, DIME, QUARTER, HALF_DOLLAR, DOLLAR);
```

The first declaration introduces a type named DAY. Just as we can say

```
COUNTER: INTEGER;
```

to declare a variable COUNTER of type INTEGER, we can say

```
TODAY: DAY;
```

to declare a variable TODAY of type DAY. Similarly, just as a variable of type INTEGER can take on integer values, a variable of type DAY can take on one of the seven values MON through SUN. In this sense we say that a type

describes a class of values.

The two types introduced above are called enumeration types. For enumeration types, the type declaration explicitly enumerates the class of values.

One of the properties of every enumeration type is that the values are ordered. In particular, the values are assumed to be enumerated in increasing order. For the type DAY, the first value is MON, the last is SUN.

For every type there are operations that can validly be performed on the values. For enumeration types, these include the relational operators for comparing values in the type. Thus if the variable TODAY has the value TUE, we may have the following expressions

```
TODAY = MON    -- value is FALSE
TODAY <= FRI   -- value is TRUE
```

The above examples also illustrate another basic idea: a programmer can introduce a type to describe a class of values needed for an application. All programmer defined types are introduced by type declarations of the form

type identifier is type definition;

The identifier specifies a name for the type. The type definition specifies the class of values and implicitly, the operations defining ways in which the values can be used. Importantly, every type definition introduces a distinct type.

With this discussion in mind, we give the basic definition of a type:

> A type characterizes a set of values and the set of operations that are applicable to the values.

In programs, all variables have an associated type. The type is that specified when the variable is declared.

One of the key issues in programming is the security with which we can draw conclusions about a program. Consider the following declarations:

```
TODAY   : DAY;
NEW_COIN: COIN;
COUNTER : INTEGER;
```

It would be meaningful to have the statements

```
TODAY    := TUE;
NEW_COIN := NICKEL;
COUNTER  := COUNTER + 1;
```

but not meaningful to have the statements

```
TODAY    := NICKEL;      -- NICKEL is not a day
NEW_COIN := TUE;         -- TUE is not a coin
COUNTER  := TODAY + 1;   -- addition is not a legal operation for days.
```

This leads us to the two basic rules for using types:

(1) A variable cannot be assigned a value of a different type.

(2) The only allowed operations on a value are those associated with the definition of its type.

These rules are strictly enforced by the language translator, and any type errors are reported during translation. We can thus draw a fundamental conclusion guaranteed by the use of types.

> The type properties declared by a programmer will not be violated during program execution.

Once the type of a variable has been declared, the programmer can refer to some of the properties of the type using the notation for predefined attributes. A predefined attribute has the form

name'attribute identifier

Returning to enumeration types, we can refer to the first and last values of the type DAY with the attributes

```
DAY'FIRST  -- the first value of the type DAY
DAY'LAST   -- the last value of the type DAY
```

In addition, the functions for computing next and previous values of the enumeration type DAY can be denoted by the function attributes:

```
DAY'SUCC  -- the successor   function for the type DAY
DAY'PRED  -- the predecessor function for the type DAY
```

Thus

```
DAY'SUCC(TUE) = WED
DAY'PRED(TUE) = MON
```

As mentioned earlier, an enumeration type is defined by enumerating its values. Such types can be used as freely as integers, and often with great clarity. For example, we may declare a table itemizing the number of hours worked on each day of the week

```
HOURS_WORKED: array (MON .. SUN) of INTEGER;
```

or equivalently

```
HOURS_WORKED: array (DAY'FIRST .. DAY'LAST) of INTEGER;
```

Further, we have a loop iterating over the days of the week

```
for CURRENT_DAY in MON .. SUN loop
   -- what to do for each value of CURRENT_DAY
end loop;
```

or equivalently

```
for CURRENT_DAY in DAY'FIRST .. DAY'LAST loop
   -- what to do for each value of CURRENT_DAY
end loop;
```

Notice the clarity of the above loops over

```
for DAY_INDEX in 1 .. 7 loop
   -- what to do for each value of DAY_INDEX
end loop;
```

Table 2.1 defines a number of enumeration types. Using such types can add considerably to the expresssiveness of a program.

Table 2.1 A Sampler of Enumeration Types

```
type DAY          is (MON, TUE, WED, THU, FRI, SAT, SUN);

type COIN         is (PENNY, NICKEL, DIME, QUARTER,
                      HALF_DOLLAR, DOLLAR);

type DIRECTION    is (NORTH, EAST, SOUTH, WEST);

type OP_CODE      is (ADD, SUB, MUL, LDA, STA, STZ);

type HALF_DAY     is (AM, PM);

type FILE_STATUS  is (OPEN, CLOSED);

type ARMY_RANK    is (PRIVATE, CORPORAL, SERGEANT, LIEUTENANT,
                      CAPTAIN, MAJOR, COLONEL, GENERAL);

type CONTROL_CHAR is (NULL, END_OF_TRANSMISSION, ENQUIRE, BELL,
                      BACKSPACE, LINE_FEED, CANCEL, ESCAPE);

type PEN_STATUS   is (DOWN, UP);

type SHAPE        is (TRIANGLE, QUADRANGLE, PENTAGON, HEXAGON);

type DRIVING_CODE is (NORMAL, LIMITED, SPECIAL, VIP);

type LETTER       is ('A', 'B', 'C', 'D', 'E', 'F', 'G',
                      'H', 'I', 'J', 'K', 'L', 'M', 'N',
                      'O', 'P', 'Q', 'R', 'S', 'T', 'U',
                      'V', 'W', 'X', 'Y', 'Z');

type HEX_LETTER   is ('A', 'B', 'C', 'E', 'F');
```

2.2 Primitive Types

In any language there are several types that are so commonly used that they are defined in the language itself. These are the primitive types. These basic types are not only useful in ther own right, but can be used to define other, more elaborate, types needed in an application.

Perhaps the simplest of all primitive types is the type named BOOLEAN. This type captures the notion of truth and falsity. It is in fact a predefined enumeration type with the type declaration

```
type BOOLEAN is (FALSE, TRUE);
```

In addition to the properties common to all enumueration types, several logical operations are defined for boolean values. These include the operators

```
and  -- logical conjunction, e.g. (TRUE and FALSE) = FALSE
or   -- logical disjunction, e.g. (TRUE or FALSE) = TRUE
not  -- logical negation,    e.g. (not TRUE) = FALSE
```

Another important enumeration type is also predefined, the type named CHARACTER. Characters are used to form messages and text. The allowed characters and their ordering are defined in a given implementation.

No language would be very useful without some facility for arithmetic computation. The most familiar arithmetic type is the type named INTEGER, used for exact arithmetic. This type denotes a finite subset of the whole numbers. The range of integer values is implicitly limited by the representation adopted by a given implemetation. The operations over this type are familiar, and includes the arithmetic operators

```
+  -- addition
-  -- subtraction
*  -- multiplication
/  -- division
```

as well as the relational operators

```
<   -- less than
<=  -- less than or equal
=   -- equal
/=  -- not equal
>=  -- greater than or equal
>   -- greater than
```

The relational operators all yield a result of type BOOLEAN.

There are two other arithmetic types, the real types. Real types provide approximations to the real numbers. With a fixed point real type the accuracy of the type is specified by giving an absolute error bound. With a floating point real type, the accuracy of the type is specified by giving a relative error bound. Consider the following declarations

```
type VOLTAGE     is delta 0.001;  -- a fixed point type
type COEFFICIENT is digits 10;    -- a floating point type
```

The type VOLTAGE denotes a set of fixed point numbers whose values have an accuracy at least as fine as 0.001. The type COEFFICIENT denotes a set of floating point numbers whose values have an accuracy of at least 10 digits. As for integers, the arithmetic and relational operators are defined for real types.

Normally, a given implementation will have one or more predefined real types, for example the types named FLOAT, LONG_FLOAT etc. These types would be likely to correspond to the arithmetic supplied by the hardware.

Before leaving the primitive types, we emphasize two fundamental points. The types BOOLEAN and CHARACTER, as well as the arithmetic types and programmer defined enumeration types, each represent a conceptual unit for describing a class of values. Furthermore, the type properties declared by the programmer remain invariant during program execution. This ensures, for example, that an integer variable always denotes an integer, and that we cannot inadvertently add a character to a real number or compare a day of the week to an integer.

2.3 Array Types

We must have ways to represent objects like a directory of area codes, a deck of cards, an instruction format, or a line of characters. These kinds of objects are different from those described earlier in this section in that their structure is important to the programmer. In particular, they have components that bear some relation to each other.

Perhaps the most familiar of such objects in programming are arrays. An array is a collection of components of the same component type.

In its simplest form an array is a description of a table. Consider the following table, which might describe the number of hours worked on each day of the week.

Day	Number of Hours Worked
Monday	8
Tuesday	8
Wednesday	8
Thursday	10
Friday	8
Saturday	0
Sunday	0

Given the enumeration type DAY declared above, this table could be declared as

```
HOURS_WORKED: array (DAY'FIRST .. DAY'LAST) of INTEGER;
```

Just as for enumeration types, array type definitions can be named in a type declaration and the name can be used to declare array variables. For example we may have

```
type WORK_DATA is array (DAY'FIRST .. DAY'LAST) of INTEGER;

HOURS_WORKED: WORK_DATA;
```

An intrinsic property of each array is that it has indices. The allowable indices are specified by giving a range of discrete values. For example, we may have the array type definitions

```
array (DAY'FIRST .. DAY'LAST) of INTEGER  -- indices are the values MON
                                          -- through SUN of type DAY

array (MON .. SUN) of INTEGER  -- another way of declaring the above array

array (1 .. 10) of INTEGER     -- indices are the integers 1 through 10
```

For any array A, the lower bound of its first index is denoted with the attribute A'FIRST, the upper bound with A'LAST.

A basic operation on arrays is indexing. It permits referencing and updating of components. For example, with the declarations

```
I: INTEGER;
A: array (1 .. 10) of INTEGER;

TODAY        : DAY
HOURS_WORKED : array (DAY'FIRST .. DAY'LAST) of INTEGER;
```

and the assignments

```
I     := 1;
TODAY := MON;
```

we may have

```
A(I)                    -- referencing the first component of A
A(I) := A(I) + 1;       -- updating the first component of A

HOURS_WORKED(MON)       -- referencing the first component of HOURS_WORKED
HOURS_WORKED(TODAY)     -- referencing the first component of HOURS_WORKED

HOURS_WORKED(TODAY) := HOURS_WORKED(TODAY) + 1;
                        -- updating the first component of HOURS_WORKED
```

In addition to indexing, assignment and equality are defined on complete arrays. The value of a complete array can be denoted by the array name. It can also be constructed by giving an aggregate. An aggregate is a listing of the values of each component. These values can be given in

positional order, or by explicitly giving each component index and its associated value.

For example, each one of the following assignments would set the value of HOURS_WORKED to the values given in the previous table.

```
HOURS_WORKED := (8, 8, 8, 10, 8, 0, 0);

HOURS_WORKED := (MON => 8, TUE => 8, WED => 8, THU => 10, FRI => 8,
                 SAT => 0, SUN => 0);

HOURS_WORKED := (MON | TUE | WED | FRI => 8, THU => 10, SAT | SUN => 0);
```

Notice that when the indices are explicitly given, the component values can be listed in any order. Notice also that array assignment requires that the number of components assigned be identical to the number declared for the array variable.

The notation for aggregates has several other forms that are especially useful for large arrays. For example, for an array

```
B: array (1 .. 100) of INTEGER;
```

the aggregate

```
(1..100 => 0)
```

denotes an array value where all components are zero. If the first four components are one, and the remaining components are zero, we may use the aggregate

```
(1 => 1,  2 => 1,  3 => 1,  4 => 1,  5..100 => 0)
```

or

```
(1..4 => 1,  5..100 => 0)
```

or even

```
(1..4 => 1,  others => 0)
```

In addition to operating on complete arrays, portions of an array can be denoted by an array slice. An array slice is a contiguous sequence of components of the first dimension of an array. For example, the first ten components of the array B above can be denoted by the slice

```
B(1..10)
```

These ten components can be assigned the values of the last ten components with the assignment

```
B(1..10) := B(91..100);
```

Arrays with unspecified index ranges

Consider the problem of writing a procedure to compute the maximum value in any array with integer components, no matter how many components are in the array. This problem is typical of cases where we want to define an array operation for which the range of the indices is immaterial.

The problem of dealing with arrays of arbitrary size can be handled by giving the type of the indices but leaving the range of the indices unspecified. Variables of such a type must, in turn, be declared with a specific index range.

Consider for example, the following declarations

```
NUM_APPLICANTS: constant INTEGER:= 20;
NUM_EMPLOYEES : constant INTEGER:= 1000;

MAX_SCORE, MAX_WAGE: INTEGER;

type ANY_INTEGER_ARRAY is array (INTEGER) of INTEGER;

SCORE_TABLE: ANY_INTEGER_ARRAY(1 .. NUM_APPLICANTS);
WAGE_TABLE : ANY_INTEGER_ARRAY(1 .. NUM_EMPLOYEES);
```

Here, ANY_INTEGER_ARRAY is an array with integer components and integer indices. The variables SCORE_TABLE and WAGE_TABLE both are of this type, but with a specifically given index range.

We can now readily write a procedure for calculating the maximum value of an array of arbitrary size.

```
procedure GET_MAX (A: in ANY_INTEGER_ARRAY;   MAX: out INTEGER) is
begin

   MAX := A(A'FIRST);

   for I in A'FIRST + 1 .. A'LAST loop
      if A(I) > MAX then
         MAX := A(I);
      end if;
   end loop;

end;
```

Notice here that the use of the attribute A'FIRST and A'LAST allow us to make use of the actual bounds of a given array argument. This procedure can thus be applied to our two arrays with the calls

```
GET_MAX(SCORE_TABLE, MAX_SCORE);
GET_MAX(WAGE_TABLE,  MAX_WAGE);
```

Strings

Strings are the basic units for describing messages and text, and thus have an important role in many applications, especially those with input and output. The type named STRING is a predefined array type with character components and integer indices. The range of the indices is unspecified, but must be equal to or greater than 1.

Since STRING is an array type, a variable of type STRING can be declared by giving the bounds of its indices. For example, we can say

```
FIRST_NAME, LAST_NAME: STRING(1 .. 10);
INPUT_LINE: STRING(1 .. 72)
```

Explicit character strings can be written in the familiar way, by enclosing the sequence of characters in quotes. For example, we can write

```
FIRST_NAME :=  "GEORGE     ";
LAST_NAME  :=  "WASHINGTON";

PUT("THIS MESSAGE");
```

Observe that an explicit character string can be viewed as a shorthand notation for a positional array aggregate. The number of components in the aggregate is the number of characters in the string.

Finally, the concatenation operator & can be applied to strings. For example, we can write

```
INPUT_LINE(1 .. 20) := FIRST_NAME & LAST_NAME;
```

2.4 Record Types

A programmer must often deal with objects having a number of different components. For example, a driving license may be an object having a

```
Driver name: a name consisting of a
   First name    : a string of 10 characters
   Middle initial: a character
   Last  name    : a string of 10 characters

License number: a nine digit integer

Expiration date: a calendar date consisting of a
   Month of year: one of the months January through December
   Day of month : an integer from 1 to 31
   Year         : an integer

Driving code: a code specifying a normal, limited, special, or vip license.
```

Depending on your driving code, there may be other information, but let's also get to that later when we discuss variant records.

The type used for groups of related objects is called a record type. A record contains a fixed number of components, each of which has a name and a value. Unlike array types, the components in a record may have different types.

For example, consider the record type definition

```
record
   FIRST_NAME     : STRING(1 .. 10);
   MIDDLE_INITIAL: CHARACTER;
   LAST_NAME      : STRING(1 .. 10);
end record
```

This type definition describes a record structure with three components, giving the three parts of a person's name. Just as for any type definition, the definition may be named in a type declaration and used in subsequent declarations. This gives rise to the following declarations, which parallel our intuitive definition of a license given above.

```
type MONTH  is (JAN, FEB, MAR, APR, MAY, JUN,
                JUL, AUG, SEP, OCT, NOV, DEC);

type NAME is
   record
      FIRST_NAME     : STRING(1 .. 10);
      MIDDLE_INITIAL: CHARACTER;
      LAST_NAME      : STRING(1 .. 10);
   end record;

type DATE is
   record
      MONTH_NAME: MONTH;
      DAY_NUM    : INTEGER;
      YEAR_NUM   : INTEGER;
   end record;

type LICENSE is
   record
      DRIVER          : NAME;
      LICENSE_NUM     : INTEGER;
      EXPIRATION_DATE: DATE;
      DRIVING_CODE    : (NORMAL, LIMITED, SPECIAL, VIP);
   end record;

PERMIT: LICENSE;  -- a variable declared with a record type
```

The basic operation on record types is component selection. For example, to refer to the driving code of PERMIT we may say

```
PERMIT.DRIVING_CODE
```

Notice that

```
PERMIT.DRIVER
```

is also a record, so that it makes sense to say

```
PERMIT.DRIVER.LAST_NAME
```

With component selection we can thus refer to the components of a record variable just as we do for referring to the components of an array. For example, we can write

```
PERMIT.LICENSE_NUM := Ø22_32_5795;

PERMIT.EXPIRATION_DATE.YEAR_NUM :=
   PERMIT.EXPIRATION_DATE.YEAR_NUM + 4;
```

Finally, just as for arrays, assignment and comparison for equality are allowed for entire record structures of identical type. For instance, if we declare

```
TODAYS_DATE, BIRTH_DATE: DATE;
```

then

```
(TODAYS_DATE = BIRTH_DATE)
```

will be true if and only if

```
(TODAYS_DATE.MONTH_NAME = BIRTH_DATE.MONTH_NAME)  and
(TODAYS_DATE.DAY_NUM    = BIRTH_DATE.DAY_NUM)     and
(TODAYS_DATE.YEAR_NUM   = BIRTH_DATE.YEAR_NUM)
```

that is, only if all corresponding components in TODAYS_DATE and BIRTH_DATE have identical values. Also, just as for arrays, aggregates can be used to denote complete record values. For example, we can assign a date to BIRTH_DATE with the positional aggregate

```
BIRTH_DATE := (JUL, 4, 1776);
```

or equivalently with the named aggregate

```
BIRTH_DATE := (MONTH_NAME => JUL, DAY_NUM => 4, YEAR_NUM => 1776);
```

Records with alternative structures

As mentioned earlier, there are cases where additional information may be required when another record component has a certain value. In our license example, the driving code may indicate a special or a limited permit requiring other information. This kind of structure is handled with a record variant. A record type with alternative structures must have a special component called a "discriminant" and a variant part giving the various substructures for possible values of the discriminant.

Consider the following revised definition of the type LICENSE:

```
type LICENSE is
   record
      DRIVER          : NAME;
      LICENSE_NUM     : INTEGER;
      EXPIRATION_DATE: DATE;
      DRIVING_CODE    : constant (NORMAL, LIMITED, SPECIAL, VIP);
      case DRIVING_CODE of
         when SPECIAL =>
            VEHICLE_TYPE: INTEGER;
            PASSENGERS   : BOOLEAN;
            ZONE_CODE    : INTEGER;
         when LIMITED =>
            CORRECTIVE_LENSES: BOOLEAN;
            DAYLIGHT_ONLY    : BOOLEAN;
            AUTO_TRANSMISSION: BOOLEAN;
         when others          => null;
      end case;
   end record;

PERMIT: LICENSE;
```

Here the component DRIVING_CODE is used as a discriminant, and the case structure

```
case DRIVING_CODE of
   when SPECIAL =>  -- components for a special license
   when LIMITED =>  -- components for a limited license
   when others  =>  -- components for other license codes
end case;
```

defines the alternative substructures. When the value of the discriminant is SPECIAL, the information for a special vehicle type is included; when its value is LIMITED, the information for a limited permit is included. Note that a variant may be specified as null, implying that there is no additional information. In particular, when the DRIVING_CODE discriminant does not have the value SPECIAL or LIMITED, the variant is explicitly stated as being empty.

The discriminant of a record must be specified as constant. This means that once a record variable is assigned a record value with a given discriminant (and thus has the alternative substucture specified by the discriminant), the discriminant of this record value cannot be changed. The record variable may, of course, be assigned a completely new record value with a different discriminant and corresponding substructure.

The components of a variant part are referenced in the same way as other components. For example, just as

```
PERMIT.LICENSE_NUM
```

refers to the license number component of PERMIT

```
PERMIT.VEHICLE_TYPE
```

refers to the vehicle type number of PERMIT. In the latter case, of course, PERMIT must have the driving code SPECIAL.

2.5 Types with Dynamic Structure

In many applications, we need to create objects during program execution and dynamically express their relationship to other objects. Such cases typically arise when we want to treat dynamically varying collections of associated objects, for example the nodes in a network, the components of a data base, the items in a linked list, or the members of a family tree. These applications can be described with access types, which provide access to dynamically created objects.

Consider, for example, the following declarations

```
type PERSON is access
   record
      NAME        : STRING(1..10);
      SEX         : (MALE, FEMALE);
      NEXT_OF_KIN: PERSON;
   end record;

ADAM, EVE, LAST_BORN: PERSON;
```

We may view an object of type PERSON as a reference (pointer, if you like) to an object with three components

(1) a string of 10 characters

(2) an enumeration value, either MALE or FEMALE

(3) a person, i.e. a reference to another object of the same type

For access types, a reference to no object is denoted by null. For example to initialize a family with no members, LAST_BORN can be initialized as a reference to no object with the assignment

```
LAST_BORN := null;
```

The creation of objects of an access type is accomplished with the new operation. Consider the statements

```
ADAM := new PERSON(NAME => "ADAM      ",
                   SEX  => MALE,
                   NEXT_OF_KIN => null);

LAST_BORN := ADAM;
```

The first statement creates a new object with the three components given by the record aggregate, and assigns to ADAM a reference to this object. The

second statement assigns the same reference to LAST_BORN. Notice that the third component of the object associated with ADAM is null.

The components of access type objects are denoted in the same way as non access type objects. For example, consider the statements

```
EVE := new PERSON(NAME => "EVE       ",
                  SEX  => FEMALE,
                  NEXT_OF_KIN => ADAM);

ADAM.NEXT_OF_KIN := EVE;
LAST_BORN        := EVE;
```

Here the statement

```
ADAM.NEXT_OF_KIN := EVE;
```

results in setting the third component of the object associated with ADAM as a reference to EVE.

Now that we have two persons in our family, we can also see the expression of dynamic relationships during program execution. The third components of ADAM and EVE now refer to each other, and LAST_BORN has been maintained as a reference to the person who was last born.

2.6 Derived Types

It is not unusual to have several distinct classes of values with identical properties and operations. For example, we may wish to describe both American dollars and Swiss francs as integers, and of course make use of the normal arithmetic properties of integers in performing our calculations. Nevertheless, dollars and francs are distinct classes of values. Like any two distinct types, we do not want to add dollars to francs, or pass a dollar amount to a procedure expecting an amount in francs. This idea is captured with the notion of derived types.

Consider for example the type declarations

```
type DOLLAR_AMT is new INTEGER;
type FRANC_AMT  is new INTEGER;
type YEAR_NUM   is new INTEGER;
```

These declarations introduce three distinct types. Each type has the same properties and values as the type INTEGER. For example we may have

```
SALARY, BONUS: DOLLAR_AMT;

SALARY := 10_000;
BONUS  := 1_000;
SALARY := SALARY + BONUS;
```

Nevertheless the three types are distinct, and as for any distinct types cannot be mixed. In particular, given

```
IMPORT_COST: FRANC_AMT;
THIS_YEAR  : YEAR_NUM;
```

we cannot have

```
SALARY := SALARY + THIS_YEAR; -- illegal addition of dollars to a year
SALARY := IMPORT_COST;        -- illegal assignment of francs to dollars
```

In general, a type named B can be derived from an existing type named A with a declaration of the form

```
type B is new A;
```

Such a declaration means that the type B has the same values, the same properties, and the same operations as A. Otherwise the two types are conceptually distinct and cannot be mixed.

Finally, the reader may have observed that strictly speaking, dollars and francs should not really be defined as integers, since it is not really correct to multiply two dollar or two franc amounts. For defining such types in their pure form, the facility for private types (see Chapter 5) can be used.

Derived types will be useful to define related types in a hierarchical fashion. For example one may want to define a type VEHICLE and later define two derived types CAR and TRUCK with the characteristics of VEHICLES. In addition, CAR and TRUCK being different types, we may have specific different procedures operating on objects of these types.

2.7 Constraints on Types

There are of course many cases where variables have a common type but where the values that a variable can denote are kept within certain limits. For instance we may know that the value of an integer variable must lie in a particular range, or that a character variable can only denote certain characters. To handle this kind of situation, we have the notion of a <u>type constraint</u>.

Perhaps the most common use of constraints is for restricting the range of values in a type. Consider the following declarations:

```
COLUMN_POS : INTEGER   range 1 .. 80;
NEXT_LETTER: CHARACTER range "A" .. "Z";
```

Here the variable COLUMN_POS is of type INTEGER, but its values are restricted to 1 through 80. The variable NEXT_LETTER is of type CHARACTER, but its values are restricted to alphabetic letters.

Range constraints can be applied to any scalar type, including user defined enumeration types and real types. For example we may have

```
WORK_DAY: DAY range MON .. FRI;
REST_DAY: DAY range SAT .. SUN;

WEIGHT: REAL range 0.0 .. 300.0;
HEIGHT: REAL range 0.0 .. 7.0;
```

In addition to making a programmer's intent clearer, the major uses of constraints is in the greater security of assignment. Constraint violations are reported by the translator when possible, or at execution time by giving an appropriate error message. A constraint specification guarantees that any inadvertant computation producing a constraint error will be detected at the point of violation.

The notion of constraints is quite general and can be applied to other basic types in the language. For records, if a previously declared record type contains several variants, a discriminant specification can be used to constrain a record variable to a specified variant. For example, you may know that your license always has a special driving code. This information can be stated as

```
MY_LICENSE: LICENSE(DRIVING_CODE => SPECIAL);
```

Here the constraint is expressed in the form of an incomplete record aggregate where values are provided only for discriminants.

While the ability to specify constraints on types is indeed useful, knowledge about the sharing of special properties should be isolated. For example, consider the variable declarations

```
LEADING_CHAR  : CHARACTER range "A" .. "Z";

MIDDLE_INITIAL: CHARACTER range "A" .. "Z";
```

which might appear in two widely separated places within a program. The two variables above have the same type and the same constraint, yet this information is repeated twice. The repetition of information is one classical source of maintenance problems, especially when one copy of the information is altered but not the other.

Just as knowledge about types can be isolated and named in a type declaration, a subtype declaration can be used to factor and name the constraints on a type. Consider the following declarations.

```
subtype LETTER is CHARACTER range "A" .. "Z";

LEADING_CHAR  : LETTER;

MIDDLE_INITIAL: LETTER;
```

The subtype LETTER serves as an abbreviation for the type CHARACTER and the constraint that the character must be an alphabetic letter. It is

important to note here that a subtype does not introduce a new type. The type denoted by a subtype is obtained by deleting all constraints given on the type or subtype name following the "is". For example, in

```
subtype LETTER     is CHARACTER range "A" .. "Z";
subtype HEX_LETTER is LETTER    range "A" .. "F";

NEXT_CHAR: HEX_LETTER;
```

the type of NEXT_CHAR is CHARACTER.

We can now show the definition of the predefined type STRING and use it to illustrate another form of constraints: index constraints.

```
subtype NATURAL is INTEGER range 1 .. INTEGER'LAST;
type STRING is array(NATURAL) of CHARACTER;
```

The subtype NATURAL characterizes natural numbers and, as stated earlier, strings are arrays of characters with bounds that are natural numbers. For objects of the type STRING the actual bounds can be specified in their declaration by an index constraint. For example

```
BUFFER: STRING(1 .. 80);
```

specifies that BUFFER is an array of type STRING; in addition the index constraint (1 .. 80) specifies that 1 is the lower bounds and 80 the upper bound.

The ability to declare and name subtypes complements the ability to declare and name types. For example consider the following subtype declarations:

```
subtype WEEK_DAY         is DAY range MON .. FRI;
subtype WEEK_END         is DAY range SAT .. SUN;

subtype SPRING           is MONTH     range MAR .. JUN;
subtype ARMY_OFFICER     is ARMY_RANK range LIEUTENANT .. GENERAL;
subtype FLOOR_NUM        is INTEGER   range 1 .. 33;

subtype SPECIAL_LICENSE is LICENCE(DRIVING_CODE => SPECIAL);
subtype VIP_LICENSE     is LICENSE(DRIVING_CODE => VIP);
```

We see here the expressiveness in using subtypes to describe knowledge about types.

The difference between types and subtypes can be summarized as follows. We use types to partition different kinds of objects into distinct classes. This partitioning is very strict, since we cannot mix objects of different types. We use subtypes to describe objects with common types, but with individual constraints on their values. Objects of different subtypes of the same type can be mixed in an expression and assigned to other variables of the same type, as long as the constraints are obeyed.

2.8 Summing Up

Having completed our tour through types, we offer a few comments. The type facility includes a number of primitive types, as well as the ability to define and name enumeration types, array types, and record types. Dynamically varying structures can be defined using access types, and new types can be derived from existing types. Moreover, a programmer can define multiple types, each building on previously defined types, and there are few conceptual limits. A number of example type definitions are given in Table 2.2.

Importantly, types allow us to develop conceptual units to model real world objects with precision and clarity. Underlying the use of types is the guarantee that the type properties declared by the programmer will not be violated during program execution. Furthermore, any constraint violations on the values of a type will be reported. While we shall discuss an even greater security of types later when we get to package modules (Chapter 5), the basic notion of types can be a powerful programming tool.

Table 2.2 Examples of Types

```
type DAY      is (MON, TUE, WED, THU, FRI, SAT, SUN);
type COIN     is (PENNY, NICKEL, DIME, QUARTER, HALF_DOLLAR, DOLLAR);
type OP_CODE is (ADD, SUB, MUL, LDA, STA, STZ);

type COEFFICIENT is digits 10 range -0.0 .. 1.0;
type VOLTAGE     is delta 0.001;

type DOLLAR_AMT is new INTEGER;
type YEAR_NUM   is new INTEGER;

subtype WEEK_DAY      is DAY       range MON .. FRI;
subtype LETTER        is CHARACTER range "A" .. "Z";
subtype CHAR_POSITION is INTEGER   range 1 .. 72;

type DATE is
   record
      MONTH_NAME: MONTH;
      DAY_NUM   : INTEGER range 1 .. 31;
      YEAR_NUM  : INTEGER range 0 .. 2000;
   end record;

type INSTRUCTION is
   record
      OPERATION: OP_CODE;
      REGISTER : INTEGER range 1 .. 12;
      ADDRESS  : INTEGER;
   end record;

type VECTOR            is array (1 .. 100)         of FLOAT;
type COEF_MATRIX       is array (1 .. N, 1 .. N)   of COEFFICIENT;
type INSTRUCTION_STACK is array (1 .. STACK_SIZE) of INSTRUCTION;

type PERSON is access
   record
      NAME       : STRING(1 .. 10);
      SEX        : (MALE, FEMALE);
      NEXT_OF_KIN: PERSON;
   end record;
```

3. EXPRESSIONS AND STATEMENTS

Every language includes constructs for calculating values, making decisions, and computing results. We turn next to these constructs, the basic expressions and statements in the language. These kinds of constructs are illustrated in the piece of open program text sketched in Example 3.1.

3.1 Expressions

An expression is a formula for computing a value. The basic elements of an expression are typified by the following examples

```
1                       -- a numeric value
TRUE                    -- a boolean value
MON                     -- a user-defined enumeration value
"ILLEGAL SYMBOL"        -- a string value
(1..10 => 0)            -- an aggregate value

INDEX                   -- the value of a variable
TEXT(I + 1)             -- the value of an array component
SYMBOL.LENGTH           -- the value of a record component

A(1..10)                -- a slice of an array
SIZE(TOKEN)             -- the value of a function call
A'FIRST                 -- the value of an attribute
```

The elements of an expression can be combined with operators. As is customary, we can use parentheses to specify the order of evaluation, or in the absence of parentheses, there are precedence levels specifying the order of evaluation. For instance, the multiplying operations * and / are applied before the adding operators + and -, the adding operators are applied before the relational operators = and <, and the relational operators are applied before the logical operators and and or.

In order to maintain the security afforded by types, each operator in an expression is applicable only to operands of specified types, and yields a result of some specified type. For example, the adding operators + and - are only applicable to numeric values of the same type and yield a result of the same type.

Example 3.1 An Open Program Text

```
INDEX := 0
SYMBOL.LENGTH := SIZE(TOKEN);
SYMBOL_FOUND  := FALSE;

while INDEX < LINE_SIZE and not SYMBOL_FOUND loop

   if INDEX + SYMBOL.LENGTH > LINE_SIZE then
      MARGIN_HIT := TRUE;
   else
      for I in 1..10 loop
         TEXT(I + 1) := SYMBOL.TEXT(I);
      end loop;
   end if;

   SEARCH_KEYS(SYMBOL.TEXT, SYMBOL_FOUND, KEY);

   if SYMBOL_FOUND then
      case KEY of
         when NUMBER     =>
            POST_VALUE(SYMBOL.TEXT);
            DELIMITER_EXPECTED := TRUE;

         when IDENTIFIER =>
            POST_SYMBOL(SYMBOL.TEXT);
            DELIMITER_EXPECTED := TRUE;
         ...
      end case;
   else
      PUT("ILLEGAL SYMBOL");
   end if;

   ...

end loop;

LINE_POS := LINE_POS + SYMBOL.LENGTH;
```

For instance with the declarations

```
type DOLLAR_AMT is new INTEGER,
type YEAR_NUM   is new INTEGER;

SALARY, BONUS: DOLLAR_AMT;
YEAR: YEAR_NUM;
```

the following expressions are valid

```
SALARY - 100       -- result is of type DOLLAR_AMT
SALARY + BONUS     -- result is of type DOLLAR_AMT
YEAR + 1           -- result is of type YEAR_NUM
```

whereas

```
SALARY + 100.0     -- 100.0 is not of type DOLLAR_AMT
SALARY + YEAR      -- different named types
```

are not.

A summary of some of the operators in Green and the type rules for their valid use are given in Table 3.1. These operators may be combined in traditional ways, as in the following expressions

```
LINE_POS + SYMBOL.LENGTH  -- simple addition
LINE_COUNT mod PAGE_SIZE  -- remainder after modulo integer division
not SYMBOL_FOUND          -- logical negation
SQRT(B^B - 4E0^A^C)       -- computation of a real value

I in 1..10                -- range test for I

INDEX < LINE_SIZE         -- boolean valued test
WARM or (COLD and SUNNY)  -- boolean valued test
```

3.2 Assignment Statements

A fundamental operation in almost every programming language is assignment, whereby the current values of a variable is replaced by a new value specified by an expression. An assignment statement has the form

```
variable := expression;
```

The variable and the expression can be of any type, including complete arrays, array slices, and complete records. However, as we recall from Chapter 2 on types, the variable and expression must have the same type. This rule guarantees that the type properties of variables are preserved during program execution.

Table 3.1 Some Basic Operators

Unary operators

Operator	Operation	Operand Type	Result Type
-	negation	numeric	same numeric type
not	logical negation	BOOLEAN	BOOLEAN

Binary operators applicable to operands of identical type

Operator	Operation	Operand Types	Result Type
+	addition	numeric	same numeric type
-	subtraction	numeric	same numeric type
mod	modulus	integer	same integer type
= /=	equality and inequality	any type	BOOLEAN
< <= > >=	test for ordering	any scalar type	BOOLEAN
and	conjunction	BOOLEAN	BOOLEAN
or	inclusive disjunction	BOOLEAN	BOOLEAN

Binary operators applicable to operands of differing type

Operator	Operation	Operand Types Left	Right	Result Type
*	multiplication	integer	integer	same integer type
		floating	floating	same floating type
		fixed	integer	same as Left
		integer	fixed	same as Right
		fixed	fixed	universal fixed
in	membership	any type	a range or subtype of the type	BOOLEAN

For example, consider the following assignments

```
INDEX    := 0;                  -- a simple initialization
TODAY    := TUE;                -- assignment of a day
DISCRIM := (B^B - 4.0^A^C);     -- assignment of a real value

SYMBOL_FOUND  := FALSE;          -- a boolean initialization
SYMBOL.LENGTH := SIZE(TOKEN);    -- assignment of a function result
TABLE(J)      := TABLE(I) + 1;   -- a table update

TEXT(1..4) := "JOHN";           -- assignment of a string value
VECTOR     := (1..10 => 0);     -- zeroing of an array
BIRTH_DATE := (JUL, 4, 1776);   -- assignment of a record value
```

Given appropriate declarations, all of these assignments are valid. Notice that with the use of suitable types, the values assigned can well reflect the real world objects they are intended to represent.

Finally, we recall that any type constraints on a variable are checked upon assignment. This guarantees that any attempted violation will be reported to the programmer.

3.3 Conditional Statements

Different situations demand different computions. The notion of conditional selection is handled by two statements, the if statement and the case statement.

The if statement allows the selection of a sequence of statements based on the truth value of one or more conditions. For example, we may have a single condition, as in

```
if (MONTH_NAME = DEC)  and  (DAY_NUM = 31) then
   MONTH_NAME := JAN;
   DAY_NUM    := 1;
   YEAR_NUM   := YEAR_NUM + 1;
end if;
```

Here the truth or falsity of the condition

```
(MONTH_NAME =  DEC)  and  (DAY_NUM = 31)
```

determines whether or not the enclosed sequence of statements is executed. Note that the parentheses are not actually needed in this condition and have only been put in for clarity.

With multiple conditions it is the first true condition that determines which sequence of statements is executed. For example, we may have a structure like

```
if    WEATHER_CONDITION = RAIN then
   -- sequence of statements describing
   -- what to do when it rains
elsif WEATHER_CONDITION = SUNSHINE then
   -- sequence of statements describing
   -- what to do when the sun shines
else
   -- sequence of statements describing
   -- what to do if it is not raining
   -- and the sun is not shining
end if;
```

More generally, a condition is any expression whose type is BOOLEAN. Thus the selecting conditions can be quite varied, as in

```
if SYMBOL_FOUND then          ...    -- use of a boolean variable
if INDEX < LINE_SIZE then     ...    -- use of a relation
if X in 1..100 then           ...    -- test for range membership
if A(1..10) = A(11..20) then  ...    -- comparison of array slices
if OCCUPIED(ROW, COL) then    ...    -- a boolean function call
```

The case statement is analogous to an if statement, except that the selection is based on the value of a single expression given at the head of the case statement.

For example, consider the case statement

```
case TODAY of
   when MON =>
      OPEN_ACCOUNTS;
      COMPUTE_INITIAL_BALANCE;

   when TUE => GENERATE_REPORT (TODAY);
   when WED => GENERATE_REPORT (TODAY);
   when THU => GENERATE_REPORT (TODAY);

   when FRI =>
      COMPUTE_CLOSING_BALANCE;
      CLOSE_ACCOUNTS;

   when SAT => null;
   when SUN => null;
end case;
```

Based on the value of TODAY of type DAY, one of the seven case alternatives will be selected. Here, as in all case statements, an alternative must be provided for each possible value of the selecting expression.

The selection values given in a when clause are not limited to single values. In particular, a listing of values or a range of values may be given. Thus the case statement above could be written

```
case TODAY of
   when MON       =>
      OPEN_ACCOUNTS;
      COMPUTE_INITIAL_BALANCE;

   when TUE..THU  => GENERATE_REPORT(TODAY);

   when FRI       =>
      COMPUTE_CLOSING_BALANCE;
      CLOSE_ACCOUNTS;

   when SAT | SUN => null;
end case;
```

Finally, there are instances where the selecting expression has multiple values and where except for a few values, the actions to be taken are identical. In these instances the choice others may be used to cover any values not given in the previous choices. For example we may have

```
case TODAY of
   when MON    => INITIALIZE_PAY_DATA;
   when FRI    => WRITE_PAYCHECKS;
   when others => null;
end case;
```

The choice others must always appear last.

3.4 Looping Statements

The basic mechanism for repeated calculations is the loop statement. The repeated calculations form the basic loop, and are always bracketed in the form

```
loop
   -- statements to be repeated
end loop;
```

A basic loop can be prefixed by an iteration specification or contain loop exit statements within the basic loop. Execution of the basic loop terminates when the iteration specification is completed or when a loop exit statement is executed.

One form of iteration specification is a for clause, which defines a loop parameter and a range of values that are sucessively assigned to the parameter on each iteration of the loop. For example, we may have the loop

```
SUM := 0;
for I in 1 .. 10 loop
   SUM := SUM + A(I);
end loop;
```

which sums ten elements of the array A.

More generally, the loop parameter can be of any discrete type, and the range of values must be given explicitly.

For example we may have

```
for DAY in MON .. FRI loop
   -- statements to be executed for successive values of
   -- DAY from MON through FRI
end loop;

for NEXT_COIN in COIN'FIRST .. COIN'LAST loop
   -- statements to be executed for successive values of
   -- NEXT_COIN from COIN'FIRST through COIN'LAST
end loop;
```

The second form of iteration specification is a while clause, which defines a condition that is tested before each execution of the basic loop. The loop is terminated when the while clause evaluates to false. For example, we may have the loops

```
while NEXT_CHAR /= END_OF_LINE loop
   SKIP_CHAR(FILE_NAME, NEXT_CHAR);
end loop;

while X < EPSILON loop
   X := F(X, Y);
   Y := G(X, Y);
end loop;
```

As mentioned above, loops may also contain explicit exit statements. For example, consider the loop

```
loop
   GET_COMMAND(DEVICE, COMMAND);
   exit when (COMMAND.STATUS = STOP);
   PROCESS_COMMAND (USER, COMMAND);
end loop;
```

This loop continues to get and process commands until the status of the input command designates a stop.

Finally, loops can be labeled, and an exit statement within the loop can be used to cause termination of the labeled loop. For example, we may have

```
<<FIND>>
   for I in 1 .. 1Ø loop
      for J in 1 .. 2Ø loop
         if A(I, J) = Ø then
            M := I;
            N := J;
            exit FIND;
         end if;
      end loop;
   end loop FIND;
```

Here both loops are thus terminated when a zero-valued array component is found.

4. PROGRAM STRUCTURE

We now turn to another central issue in program construction, the composition of program units. A program unit characterizes a logical unit that is meaningful to the programmer. There are three forms of program units: subprograms, package modules, and task modules.

In this chapter we discuss the general rules for organizing program units, as well as the specific properties of subprograms. The specific properties of package modules and task modules are discussed in the following two chapters.

4.1 Subprograms

We are all familiar with the notion of a subprogram, and could hardly imagine a programming language without some such feature. Subprograms allow the programmer to package often elaborate computations and parameterize their behavior.

There are two forms of subprograms, procedures and functions. Procedure subprograms allow the definition of a sequence of actions; procedures are invoked with a subprogram call statement. Function subprograms allow the definition of a sequence of computations returning a value; functions are invoked with a subprogram call within an expression. Side-effects, i.e. assignments to global variables, are not allowed within functions.

Consider the following subprogram, given earlier in Section 1.4:

```
procedure COMPUTE_RISE(V_X,V_Y,DISTANCE: in REAL;  RISE: out REAL) is
   TIME : REAL;
begin
   TIME := DISTANCE/V_X;
   RISE := V_Y*TIME - (G/2.0)*(TIME**2);
end;
```

This procedure has the following general form

```
procedure COMPUTE_RISE (parameters) is
   -- local declarations
begin
   -- local statements
end;
```

The parameters of a procedure each have a mode, defined as follows

(a) **in** parameters, for example V_X, V_Y, and DISTANCE, which act as logical inputs

(b) **out** parameters, for example RISE, which act as logical outputs

(c) **in out** parameters, which act as input variables whose values may be updated during execution of the procedure.

If no mode is specified, the mode **in** is assumed. Within a subprogram the value of an **in** parameter may not be updated.

The name of a procedure and its parameter specifications identify the appearance of the procedure to the user. The names of the parameters are considered local to the body of the procedure. Importantly, a procedure can contain local declarations describing internal information. The statements within the body of the procedure describe the internal behavior needed to achieve the desired effect.

In form, a function subprogram is similar to a procedure subprogram, but as mentioned above is used to compute a value for use within an expression. Consider the following function subprogram, which is based on our previous example.

```
function NET_RISE(V_X, V_Y, DISTANCE: REAL) return REAL is
   TIME, RISE: REAL;
begin
   TIME := DISTANCE/V_X;
   RISE := V_Y*TIME - (G/2.0)*(TIME**2);
   return RISE;
end;
```

This function has the following general form

```
function NET_RISE (parameters) return type of result is
   -- local declarations
begin
   -- local statements
end;
```

Observe that the type of the value returned by a function must be specified, and that the function body must include one or more return statements. Execution of a return statement terminates execution of the subprogram and specifies the value to be returned. A function can only have **in** parameters, and the only permitted assignments are to variables declared within the function body. These restrictions guarantee that the computation of a returned value will not change any external values.

Finally, we can also write a kind of in line subprogram called a block. A block is a statement that acts as a parameterless procedure implicitly called at the place where it is defined. For example, we may have the block

```
declare  -- compress blanks
   LINE_PTR, CHAR_PTR : INTEGER;
begin
   -- statements
end;
```

Like a procedure, a block can introduce local declarative information.

4.2 Calling Subprograms

Once a subprogram has been defined, we want to use it, usually from many places in a program. This is accomplished with subprogram calls.

The most familiar form of a subprogram call uses a positional notation where the name of the subprogram is followed by a list of arguments, one for each parameter and in the same order as the parameters. For our procedure COMPUTE_RISE we might have the subprogram call statements

```
COMPUTE_RISE(X_VELOCITY, Y_VELOCITY, TARGET_DISTANCE, RISE);

COMPUTE_RISE(50.0, 60.0, EDGE_POSITION + 10.0, Y_VALUE);
```

In each case an argument must have the same type as the corresponding parameter. This rule follows naturally from our view of types and guarantees that a procedure will not be called with arguments of incompatible types.

As illustrated in the above calls, an argument corresponding to an **in** parameter can be an expression. For **out** and **in out** parameters, the argument must be a variable; the value of such a variable may be updated by execution of the procedure.

In many cases, simply listing the arguments of a subprogram does clarify their logical roles. For this purpose we can explicitly state the association between arguments and their corresponding parameters. For example, consider the procedure specification

```
procedure MERGE(NEW_ITEMS: in TABLE;  ITEMS_TO_UPDATE: in out TABLE);
```

Instead of a positional call, say

```
MERGE(SOURCE_DATA, LINE_DATA);
```

we may have

```
MERGE(NEW_ITEMS := SOURCE_DATA,  ITEMS_TO_UPDATE :=: LINE_DATA);
```

In such calls the arguments can be given in any order. For example, we may also have

```
MERGE(ITEMS_TO_UPDATE :=: LINE_DATA,  NEW_ITEMS := SOURCE_DATA);
```

Notice that the correspondence between an argument and its parameter also indicates its logical role as an input or output. In particular, we have

```
parameter :=  value       -- for in parameters
parameter  =: variable    -- for out parameters
parameter :=: variable    -- for in out parameters
```

Where long parameter lists are common and have standard default values, as in general utility packages, such calls can provide especially high readability. In particular, a default value can be specified for an in parameter by giving the default value in the subprogram specification. When the default value is to be used in a call, the corresponding argument can be omitted.

For example, consider the procedure specification

```
procedure GENERATE_REPORT(DATA_FILE  : in FILE     := SUPPLY_DATA;
                          NUM_COPIES : in INTEGER  := 1;
                          HEADER     : in LINE     := STND_HEADER;
                          CENTERING  : in BOOLEAN  := TRUE;
                          PAGE_LOC   : in POSITION := TOP_RIGHT;
                          TYPE_FONT  : in PRINTER  := NORMAL)
```

All of the following calls are acceptable

```
GENERATE_REPORT;

GENERATE_REPORT(NUM_COPIES := 5);

GENERATE_REPORT(DATA_FILE := EMERGENCY_DATA);

GENERATE_REPORT(HEADER     := "FINAL STATUS REPORT",
                PAGE_LOC   := TOP_CENTER,
                TYPE_FONT  := SPECIAL,
                NUM_COPIES := 100);
```

Each call above makes use of the default values supplied in the specification. Notice again that the arguments can be given in any order.

Finally, there are procedures where certain arguments are always present, usually in a standard order, and the remaining arguments are optional. In these cases the initial arguments can be given positionally, and any remaining arguments can be listed with their associated parameters. For example, consider the procedure specification

```
procedure PLOT(X, Y   : in REAL;
               PEN    : in PEN_POSITION := DOWN;
               GRID   : in BOOLEAN      := FALSE;
               ROUND  : in BOOLEAN      := FALSE)
```

The following calls may be given

```
PLOT (0.0, 0.0);

PLOT (X_MIN, Y_MIN, GRID  := TRUE);

PLOT (X_MAX, Y_MAX, ROUND := TRUE, PEN := UP);
```

4.3 Separation of Program Unit Bodies

All program units are characterized by two parts

- An interface, which summarizes its properties to the user
- An implementation, which describes its internal behavior

We discuss here the ability to separate the interface of a program unit from its implementation. In particular, every program unit can be defined in two textually separate parts

- its declaration, which specifies its interface
- its body, which specifies its implementation

The ability to separate a declaration from its body provides a single basis for achieving several important programming objectives. We can enhance program readability, since we can consolidate the declarations of relevant program units with other declarative information. Furthermore, we can isolate the description of their bodies, which often are spread over many pages of text. Finally, this ability is essential for modules (Chapters 5 and 6) and separate translation (Chapter 9) to be discussed later.

For subprograms, a declaration gives the name of the subprogram, its parameters, and the type of any returned value. The body gives the local declarations and statements.

For example, consider the following sequence of declarations.

```
procedure ADD      (I: in ITEM;  Q: in out QUEUE);
procedure REMOVE   (Q: in out QUEUE);
   ...
function  FRONT    (Q: QUEUE) return ITEM;
function  IS_EMPTY(Q: QUEUE) return BOOLEAN;
```

These declarations specify procedures for adding an item to a queue and for removing the item in the front of the queue from the queue, as well as functions for obtaining the front item from a queue and testing if the queue is empty. Collectively, these subprograms define a set of operations for working with queues, and their declarations can accordingly be grouped as above.

Notice that the calling conventions for the four subprograms are collected in a single place. If these declarations were not grouped (as is the case when we give only the subprogram bodies), we might have to scan many pages of text to obtain this information.

The bodies of these subprograms can be given after all declarations, and have the form

```
procedure ADD(I: ITEM; Q: in out QUEUE) is
   ...
end;

   ...

function IS_EMPTY(Q: QUEUE) return BOOLEAN is
   ...
end;
```

Notice that the parameters of each subprogram are repeated in the body. For readability, this is required. Finally, since a subprogram body thus includes the information in a subprogram declaration, the declaration need not be given unless the subprogram is called in a previously given body. This is the case for the subprograms given earlier in the text. Note however, from a readability point of view, we can always separate a procedure declaration from its body.

4.4 Overloading of Subprograms

There are situations where we want to define the same conceptual operation on arguments of different types. A classical case is a print operation for printing different types of values.

Consider the procedure declarations

```
procedure PUT(X: INTEGER);
procedure PUT(X: REAL);
procedure PUT(X: STRING);
```

for respectively printing the string representation of an integer, floating point number, or string.

The bodies of each procedure will differ, since they are dependent on the format for printing integers, floating point numbers, and strings. The use of two or more subprograms with the same name but different types of parameters is called overloading.

Overloaded subprograms can be called in the conventional manner, for example

```
PUT(I + 1);            -- call for printing an integer.
PUT(SQRT(Y));          -- call for printing a floating point number.
PUT("THIS MESSAGE");   -- call for printing a string.
```

The key idea here is that the choice of the corresponding subprogram body is based on using the subprogram with the appropriate parameter types. Accordingly, since each of the above calls matches one (and only one) of the declarations of PUT, the appropriate body can be determined. We note in passing that this use of overloading is similar to the use of + as an operator both for integer addition and floating point addition.

4.5 Nesting and Visibility

We observe that subprograms can contain declarative parts, which may in turn contain declarations and bodies of other subprograms. This textual embedding of program units is called nesting.

Nesting serves several programming objectives. Most importantly, it allows us to provide a logical organization for program units. An analogy may be made with an encyclopedia, where the materials are organized into subjects and subsubjects. Accordingly, related units can be grouped and maintained together, at the level where they are needed.

Nesting also allows us to give local names for newly declared objects, independently of any outer use of the names. Thus the programmer can devise names for items without the need to keep the names of objects at different levels distinct. In programs with a large number of names, this convenience is especially great. Finally, nesting allows us to share the visibility of common information given in outer declarative parts.

Consider, for example, the skeleton procedure of Example 4.1. Here the procedures COMPUTE_SCALING_DATA and SCALE_VALUES are nested within the procedure PLOT_QUADRATIC_EQUATION. This example gives rise to the two fundamental rules about nesting:

- Items declared within a unit are local to the unit.
- A reference to an item not declared within a unit refers to the item declared in the most immediately enclosing unit.

In our example, the items declared within the procedure COMPUTE_SCALING_DATA are independent from declarations given in the main procedure or in the other nested procedure. Furthermore, both nested procedures share visibility of information given in the main procedure, for example the constant NUM_POINTS or the type GRID_ARRAY.

Example 4.1 Nesting of Procedures

```
procedure PLOT_QUADRATIC_EQUATION (A, B, C: in REAL) is

   -- This procedure produces a graphical plot of the equation
   -- Y = A*(X**2) + B*X + C

   NUM_POINTS : constant INTEGER := 100;
   type GRID_ARRAY is array (1..NUM_POINTS) of REAL;

   -- other declarations

   procedure COMPUTE_SCALING_DATA(COORDINATES  : in  GRID_ARRAY;
                                  MIN_VALUE    : out REAL;
                                  MAX_VALUE    : out REAL;
                                  SCALE_FACTOR : out REAL) is
      -- local declarations
   begin
      -- computation of MIN_VALUE, MAX_VALUE, and SCALE_FACTOR
   end;

   procedure SCALE_VALUES (SCALE_FACTOR : in     REAL;
                           COORDINATES  : in out GRID_ARRAY) is
      -- local declarations
   begin
      -- scaling of COORDINATES
   end;

begin
   -- main routine for plotting the equation
end;
```

Notice that as a result of nesting, a program unit may declare a new meaning for an identifier that happens to be also declared in an outer unit. Within the nested unit, the identifier is associated with new meaning. Thus the meaning associated with an identifier is always given by the innermost available declaration of the identifier.

4.6 Restricting Visibility

In large programs we will, of course, have many program units. The organization of program units is thus critical to the writing and maintenance of such a program. In Chapter 9 we discuss the notion of separate translation of program units, an important factor in writing any large program. Here we discuss the notion of restricting the accessibility of names within a single compilation unit. This notion is also central to the notion of separate translation.

In a nested program unit, we may want to guarantee that the unit has limited visibility to information in outer units. On the other hand, in order to identify the net effect of a program unit, we may want to state explicitly any dependency on information in an outer unit. These dual purposes are handled with the notion of a visibility restriction.

Consider the program sketch of Example 4.2. Here we have several levels of nesting. The nested procedure DISPLAY_DATE is marked as restricted:

```
restricted
procedure DISPLAY_DATE (parameters) is
   ...
end;
```

As a result, the only names that are visible from outer units are those in the predefined environment for the program. Thus any names declared in an outer unit have no effect on this procedure. The effect of the visibility restriction is thus to restart with a completely fresh set of names, just as for the main procedure.

A visibility restriction need not always be as severe as the above case. In particular, it can include the name of an enclosing unit, as in

```
restricted (DISPLAY_TEXT_ITEM)
procedure DISPLAY_SECTION_HEADER(parameters) is
   ...
end;
```

EXAMPLE 4.2 RESTRICTED PROGRAM UNITS

```
procedure FORMAT_TEXT is
   ...
   procedure RIGHT_SHIFT(parameters) is
      ...
   end;

   procedure LINE_TAB(parameters) is
      ...
   end;

   procedure DISPLAY_TEXT_ITEM(parameters) is
      ...
      restricted(DISPLAY_TEXT_ITEM)
      procedure DISPLAY_SECTION_HEADER(parameters) is
         ...
      end;

      restricted
      procedure DISPLAY_DATE(parameters) is
         ...
      end;
      ...
   end;
   ...
end FORMAT_TEXT;
```

In this case, in addition to the predefined names, the names declared in each enclosing unit up to DISPLAY_TEXT_ITEM are also visible in the procedure.

A visibility restriction can also include the names of modules (see Chapter 5) and thus selectively provide access to the information given in the modules. In all cases, the purpose of the visibility restriction is the same: to control the set of identifiers that are visible within a program unit.

4.7 Summary

This completes our second level of introduction to Green. The facilities described thus far have analogy in many other languages, and consequently can be used for a wide range of programming applications.

In the following sections we describe facilities demanded by many specialized but critical applications. Often, we expand on the facilities already described to meet these needs.

5. PACKAGING DATA AND SUBPROGRAMS

The notion of packaging is a central issue in the construction of programs. Subprograms provide one important form of packaging. In particular, they allow the description of a (sometimes complex) computation that can be characterized by a simple interface to the user. As such, they provide a basic unit in program decomposition and allow for the orderly development of both small and large programs.

In this section we describe another important form of packaging, the package module. We shall see that package modules can cover a wide range of uses, including named collections of common data, groups of related subprograms that collectively carry out some activity, and definitions of private types with specialized operations. Generally speaking, a package module provides a service. Like any service, it may rely on internal data and services that are not (and should not be) taken advantage of by the user.

5.1 The Visible Information

Every package module is specified by giving a list of declarative information. This declarative information can be made accessible to a user, and is called its visible part.

Consider, for example, the following package module:

```
package METRIC_CONVERSION_FACTORS is
   CM_PER_INCH: constant FLOAT := 2.54;
   CM_PER_FOOT: constant FLOAT := 30.48;
   CM_PER_YARD: constant FLOAT := 91.44;
   KM_PER_MILE: constant FLOAT := 1.609;
end;
```

This simple package module defines a collection of four constants. We can readily imagine providing a more complete list of such constants, including those for areas and volumes as well as those for weights. Isolating common declarative information in a package module not only allows other program units to make use of the information, but can guarantee that the units use the same names for the constants and the same values.

The above example illustrates the general form of all package modules that only have a visible part.

```
package METRIC_CONVERSION_FACTORS is
   -- visible information
end;
```

The visible information is generally given as a sequence of declarations.

Even within this simple framework, there are many useful variations. For example, we may group common variables into a package module.

```
package WEATHER_DATA is
   ALTITUDE   : array (-90 .. 90, -90 .. 90) of FLOAT;
   TEMPERATURE: array (-90 .. 90, -90 .. 90) of FLOAT;
   WIND_SPEED : array (-90 .. 90, -90 .. 90) of FLOAT;
end;
```

Just as for constants, other program units can make use of this common data, for example to read, update, or analyze the arrays.

And what about types? In a typed language, groups of constants and variables are likely to include the declaration of related types. For example we may have

```
package GOLF_INFORMATION is
   type GOLF_CLUB   is (DRIVER, IRON, PUTTER, WEDGE, MASHIE);
   type GOLF_SCORE  is new INTEGER;
   type HOLE_NUMBER is new INTEGER range 1 .. 18;
   type HANDICAP    is new INTEGER range 1 .. 9;
   type SCORE_DATA  is array (1 .. 18) of GOLF_SCORE;

   PAR_VALUE      : constant SCORE_DATA :=
                    (5, 3, 4, 4, 3, 4, 5, 4, 4,
                     3, 4, 4, 4,5, 3, 4, 4, 5);
   PAR_FOR_COURSE: constant GOLF_SCORE := 72;

   TOTAL_SCORE    : GOLF_SCORE;
end;
```

This package module not only defines common constants and variables, but defines the type of their values. Thus any units that make use of the module will also have access to the relevant types. Importantly, a user can declare constants and variables of these types, as in

```
NEXT_HOLE   : HOLE_NUMBER;
MY_SCORE    : GOLF_SCORE;
MY_HANDICAP: constant HANDICAP := 10;
```

In all of the above examples, the effect is the same: a package module specifies a collection of logically related information that can be made accessible to other program units.

5.2 Making Use of the Visible Information

While a package module defines a collection of information, the information is not directly visible to other program units.

One of the ways to refer to an identifier declared in a package module is with a name in the dot notation form

module_name.declared_identifier

For example, consider the package module WEATHER_DATA introduced earlier. The array WIND_SPEED could be denoted with the name

```
WEATHER_DATA.WIND_SPEED
```

and thus we might have the assignment

```
WEATHER_DATA.WIND_SPEED(I,J) := WIND_READING;
```

Alternatively, all of the information in a visible part can be made directly visible in another program unit by giving a use clause at the beginning of the unit. For example, we may have the subprogram

```
procedure UPDATE_WEATHER_DATA is

   use WEATHER_DATA;
   I,J: INTEGER range -90 .. 90;
   ALT_READING, TEMP_READING, WIND_READING: FLOAT;

begin

   GET_ZONE_COORDINATES(I,J);
   GET_SENSOR_VALUES(ALT_READING, TEMP_READING, WIND_READING);

   ALTITUDE(I,J)     := ALT_READING;
   TEMPERATURE(I,J)  := TEMP_READING;
   WIND_SPEED(I,J)   := WIND_READING;

end;
```

The effect of the use clause is to make the identifiers declared in WEATHER_DATA visible just as if they were declared at the point of declaration of the package WEATHER_DATA.

We see above that the information in a package module is made visible on a selective basis. Any program unit that has need of the information in a module can make the information visible; the other units are unaffected. Thus the identifiers declared in the visible part do not flood the name space of the entire program. In a large programming system where there might be many such modules (each offering visibility to certain information), the control and selective use of the information can provide a high degree of security.

Finally, recall that the effect of a use clause is to make identifiers visible just as if they were declared at the point of the module declaration. As a consequence, an item may happen to be hidden by an inner declaration of the same identifier. In such a case, the item declared in the package module can still be denoted with the dot notation.

5.3 Providing Operations Over Data

In our previous examples, the package modules provided a certain service, namely access to information. Data alone may not be enough. We may want to offer operations to transform the data.

As mentioned earlier, the visible part of a package module generally contains a sequence of declarations. Importantly we recall that a subprogram can be given in two separate parts: its declaration and its body. This gives rise to the second basic form of package modules, in which the visible part not only contains the declaration of constants, variables, and types, but also the declaration of related subprograms. The behavior of these subprograms is described separately, in a "package module body".

Consider the problem of defining a table management package for inserting and retrieving items. The items are inserted into the table as they are posted. Each posted item has an order number. The items are retrieved according to their order number, where the item with the lowest order number is retrieved first.

From the user's point of view, the package is quite simple. There is a type called ITEM designating table items, a procedure INSERT for posting items, and a procedure RETRIEVE for obtaining the item with the lowest order number. There is a special item NULL_ITEM that is returned when the table is empty.

A sketch of the visible part of such a package is given as follows:

```
package TABLE_MANAGER is
   type ITEM is
      record
         ORDER_NUM: INTEGER;
         ITEM_CODE: INTEGER;
         ITEM_TYPE: CHARACTER;
         QUANTITY : INTEGER;
      end record;

   NULL_ITEM: constant ITEM :=
      (ORDER_NUM | ITEM_CODE | QUANTITY => 0, ITEM_TYPE => " ");

   procedure INSERT  (NEW_ITEM  : in  ITEM);
   procedure RETRIEVE(FIRST_ITEM: out ITEM);
end;
```

To complete the above package we need to define the implementation of the two procedures INSERT and RETRIEVE. This is achieved by defining the two procedure bodies within a module body. The module body serves the same purpose as a subprogram body. In particular, it defines the internal details needed to implement the interface provided to the user. Like subprogram bodies, the characteristics of the module bodies are not visible to the user.

A module body defining the implementation of the table management package is sketched in Example 5.1. The details of implementing such packages can be quite complex, in this case involving a two-way linked table of internal items. A local housekeeping procedure EXCHANGE is used to move an internal item between the busy and the free lists. Two local functions, FREE_LIST_EMPTY and BUSY_LIST_EMPTY, as well as local constants and variables are also used. The initial table linkages are established in an initialization part.

We see here the general form of a package module whose visible part contains subprogram declarations.

```
package TABLE_MANAGER is
   -- visible information
end;

package body TABLE_MANAGER is
   -- bodies of visible subprograms, plus any internal
   -- information needed to implement the bodies
begin
   -- statements initializing any internal data
end;
```

While in essence the module body provides the bodies for any visible subprograms, writing such bodies may in turn require the use of local data, local types, and local subprograms. The use of local data may in turn require the execution of statements to initialize their values. These statements are executed when the module body itself is elaborated.

In general, the module body implements the service promised to the user. As before, only the visible part of the package is exposed to the user.

Example 5.1 Module Body for a Table Management Package

```
package body TABLE_MANAGER is

   SIZE: constant INTEGER := 2000;
   subtype INDEX is INTEGER range 0 .. SIZE;

   type INTERNAL-ITEM is
      record
         CONTENT   : ITEM;
         NEXT_ITEM: INDEX;
         PREV_ITEM: INDEX;
      end record;

   TABLE: array (INDEX) of INTERNAL_ITEM;

   FIRST_FREE_ITEM: INDEX := 0;
   FIRST_BUSY_ITEM: INDEX := 1;

   function FREE_LIST_EMPTY return BOOLEAN is
      ...
   end;

   function BUSY_LIST_EMPTY return BOOLEAN is
      ...
   end;

   procedure EXCHANGE(FROM,TO: in INDEX) is
      ...
   end;

   procedure INSERT(NEW_ITEM: in ITEM) is
      ...
   end;

   procedure RETRIEVE(FIRST_ITEM: out ITEM) is
      ...
   end;

begin

   -- code for initialization of the table linkages

end TABLE_MANAGER;
```

5.4 Private Information

Although the mechanism ilustrated above provides a simple and rich facility for abstractions, it does not guarantee the integrity of these abstractions. In our previous example, nothing prevents a user from accessing the components of an item and changing them arbitrarily. This may be convenient, but it may also lead to serious problems, especially in applications where we wish to prevent users from getting their hands on the data. Thus some safeguards are necessary if we wish to ensure the integrity of information. This brings up the issue of private information, in particular the use of objects whose type is specified as private.

Consider the following package.

```
package SIMPLE_INPUT_OUTPUT is
   type FILE_NAME is private;

   procedure OPEN  (F: out FILE_NAME);
   procedure READ  (ITEM: out INTEGER;  F: in FILE_NAME);
   procedure WRITE (ITEM: in  INTEGER;  F: in FILE_NAME);

private
   type FILE_NAME is new INTEGER range 0 .. 50;
end;


package body SIMPLE_INPUT_OUTPUT is
   -- implementation of file handling procedures
end;
```

As before, a user of this package module can declare variables of type FILE_NAME, and can pass these variables to the procedures OPEN, READ, and WRITE. However, because the type FILE_NAME is declared as private, the user cannot make use of the fact that this type is internally defined as an integer, that is, the user cannot perform arithmetic on variables of type FILE_NAME. The only language defined operations that can be used are assignment and comparison for equality or inequality. If we wish, even these operations can be forbidden by declaring FILE_NAME as restricted, as in

```
restricted type FILE_NAME is private;
```

Observe that the above package module has the form

```
package SIMPLE_INPUT_OUTPUT is
   type FILE_NAME is private;
   ...
private
   type FILE_NAME is type definition;
end;
```

```
package body SIMPLE_INPUT_OUTPUT is
   ...
end;
```

Here the visible part of the package module is followed by a "private part", in which the type definition of each private type is given. This form is characteristic of any case where we want complete control over the operations of a type. Such package modules serve a dual role. On the one hand, they prevent a user operating on data of the type defined in the module. On the other hand, they implement the notion of an "abstract" data type, where the only operations over the type (aside from assignment and equality) are those given in the module.

5.5 A Note on Restricted Program Units

As mentioned in Chapter 4 on program structure, a visibility restriction can be given for a program unit, and thus limit the visibility of the unit to names declared in an enclosing unit.

In addition, a visibility restriction can include the names of modules outside the enclosing unit. If included, these module names are also visible in the restricted unit, and thus can be used in selected components and use clauses. The purpose of a visibility restriction, to state any dependencies on other units, is thus carried over to modules. This ability is not only useful in its own right, but is needed for separate translation of modules, as discussed in Chapter 9.

5.6 Summary

We thus see that package modules, like subprograms, offer a powerful tool for abstraction. The services they provide range over a broad continuum, marked by three basic capabilities

- the declaration of named collections of data and types
- the definition of packages of related subprograms
- the declaration of data whose type properties are private.

Furthermore, with the dot notation and use clauses, a user can provide selective access to the visible information.

6. PARALLEL PROCESSING

We are all familiar with events that can take place concurrently with other, perhaps related, events. The operation of several moving trains on a rail network, the handling of multiple lines of customers at a bank, and the simultaneous operation of multiple devices in a computing system are typical examples. The facilities for parallel processing provide a conceptual framework for dealing with such problems.

Our discussion here will make frequent use of a single example. We wish to write a program to decode messages. Let us not worry about what the messages mean. They are generated at some remote field station, decoded, and then printed on a line printer. In particular, we wish to define three program units, named GENERATE_MESSAGES, DECODE, and PRINT_MESSAGES.

- GENERATE MESSAGES: This program unit reads data from several sources, and as a result generates encoded messages on a character by character basis.

- DECODE: This program unit receives characters, decodes them according to some formula, and transmits the decoded characters.

- PRINT_MESSAGES: This program unit receives characters, and when it obtains a full line of text, displays the line on a line printer.

The important point about our program is that the three program units are conceptually independent and can progress at their own rates. Whether the statements in one program unit are executed before, at the same time , or after those in another unit is most of the time of no particular concern. It only matters when they must synchronize (for example a message must be generated before it is decoded). Apart from such interactions the three program units can operate concurrently.

6.1 Textual Appearance of a Task

A task is a program unit that can be executed concurrently with other tasks. In form, a task is very similar to a package module. Indeed, both a task module and a package module provide an encapsulation of a service. The major difference between the two is that the body of a task module is not executed until the task is explicitly initiated, and then both the initiating and initiated tasks can operate concurrently. A task module can also contain entries. Entries externally look like procedures, but when entries are called they specify synchronization and communication between the calling and the called tasks.

Consider the following sketch of the task module for decoding characters.

```
task DECODE is
   entry SEND_CHAR    (C: in  CHARACTER);
   entry RECEIVE_CHAR(C: out CHARACTER);
end;

task body DECODE is
   -- local declarations
begin
   -- statements for decoding characters
end;
```

This task has two entries, SEND_CHAR and RECEIVE_CHAR. Just as for a package module with two visible procedures, another task can issue a call to one of these entries by means of a normal procedure call statement. The behavior of the task as well as the effect of an entry call is specified in the module body. We defer the details of the body for a little later.

A task can have an empty visible part, that is, offer no service to other units. Such a task can, of course, be executed concurrently with other tasks. It can also utilize the services of other tasks. Consider the task modules GENERATE_MESSAGES

```
task GENERATE_MESSAGES;

task body GENERATE_MESSAGES is
   -- local declarations
begin
   -- statements for generating character codes
end;
```

and PRINT_MESSAGES

```
task PRINT_MESSAGES;

task body PRINT_MESSAGES is
   -- local declarations
begin
   -- statements for printing characters
end;
```

These tasks have no visible part, but can be executed concurrently with each other and the task DECODE. As we shall see, they can also make calls to the entries of the task DECODE.

6.2 Organization of Multiple Tasks

We recall from our discussion of package modules that on entrance into a program unit containing a package module, the module body (if any) is executed at once in order to initialize the module. Thereafter the package module is purely passive in the sense that no further statements are

executed, unless one of its visible subprograms is explicitly called.

On the other hand, on entrance into a program unit containing a task module, the module body is not executed until specifically indicated with an initiate statement. Thereafter the task module is active in the sense that it can continue execution in parallel with the initiating task and other initiated tasks.

Thus to organize multiple taks we need to declare them in a program unit and initiate their execution. Consider the following sketch, that of our program for decoding messages.

```
procedure MESSAGE_DECODING is

   task GENERATE_MESSAGES;

   task DECODE is
      entry SEND_CHAR    (C: in  CHARACTER);
      entry RECEIVE_CHAR(C: out CHARACTER);
   end;

   task PRINT_MESSAGES;

   -- bodies for the tasks

begin

   initiate GENERATE_MESSAGES, DECODE, PRINT_MESSAGES;

end MESSAGE_DECODING;
```

Execution of an initiate statement results in execution of the bodies of each named task. The bodies can be executed in parallel with each other and with the body of the initiating procedure. The main program, here the initiating procedure MESSAGE_DECODING, is implicitly considered to be a task. In our case we have three concurrent tasks in addition to the main procedure.

As far as termination is concerned, a task will normally terminate on reaching its final end. When the unit containing the declaration of local tasks reaches its end, it will wait until all locally declared tasks have terminated unless, of course, they have already done so. In our example above, the main procedure will wait at its end for the three tasks GENERATE_MESSAGES, DECODE, and PRINT_MESSAGES to terminate.

In the above discussion we have treated the notion of several tasks executing in parallel. This is essentially a conceptual viewpoint. Whether tasks are physically executed in parallel depends on the underlying implementation. In a system with multiple processors, actual parallel execution may occur. In a system with a single processor, only one task can really be active at a given time.

6.3 Communication between Tasks

In any system with related tasks, there must be some form of communication. We clearly do not want the trains on a rail network to collide, we may want to ensure that two bank tellers do not make conflicting transactions on the same account, or we may need to coordinate the actions of the devices in a computing system.

The basic form of communication between tasks is through calls to an entry. Consider the module body for the task GENERATE_MESSAGES

```
task body GENERATE_MESSAGES is
   NEXT_CODE: CHARACTER;
begin
   loop
      -- code for receiving data and
      -- generating NEXT_ CODE

      DECODE.SEND_CHAR(NEXT_CODE);  -- entry call
   end loop;
end;
```

and for the task PRINT_MESSAGES

```
task body PRINT_MESSAGES is
   NEXT_CHAR: CHARACTER;
   INDEX    : INTEGER;
   LINE     : STRING(1 .. 120);
begin
   INDEX := 1;
   loop
      DECODE.RECEIVE_CHAR(NEXT_CHAR);  -- entry call
      LINE(INDEX) := NEXT_CHAR;
      if INDEX < 120 then
         INDEX := INDEX + 1;
      else
         PRINT(LINE);
         INDEX := 1;
      end if;
   end loop;
end;
```

Both of these modules loop forever, respectively generating codes and printing text. (Note: we shall put a stop button in later).

Here we also have the two entry calls

```
DECODE.SEND_CHAR(NEXT_CODE);

DECODE.RECEIVE_CHAR(NEXT_CHAR);
```

As we see, an entry call has the same form as a procedure call. The difference lies in their internal behavior. For procedures, the corresponding actions (given by the procedure body) are executed when the

procedure is called and the actions are executed immediately. For entries, the corresponding actions (given by an accept statement) are executed by the task module containing the entry. Moreover, the actions are only executed when the called task is prepared to accept the entry call. Thus the calling and the called tasks may be considered as meeting in a rendezvous.

To illustrate this behavior, we complete our example with the module body for DECODE.

```
task body DECODE is
   CODE, CHAR: CHARACTER;
begin
   loop
      accept SEND_CHAR(C: in CHARACTER) do
         CODE := C;
      end;

      -- statements for decoding the value of CODE
      -- and producing the decoded value in CHAR

      accept RECEIVE_CHAR(C: out CHARACTER) do
         C := CHAR;
      end;
   end loop;
end;
```

This module also loops forever, repeatedly obtaining a code via an entry call to SEND_CHAR, applying the decoding algorithm, and transmitting the decoded value via a call to RECEIVE_CHAR.

Our interest here lies in the accept statements corresponding to entry calls. An accept statement has the partial appearance of a procedure body. The accept statement repeats the formal part of the entry specification, which is followed by the statements to be executed during the rendezvous. These are delimited by **do** and **end** and are the scope in which the parameters of the entry are accessible.

There are two possibilities for a rendezvous, according to whether the calling task issues a calling statement such as

```
DECODE.SEND_CHAR(NEXT_CODE);
```

before or after a corresponding accept statement is reached by the called task DECODE. Whichever gets there first waits for the other. When the rendezvous is achieved the appropriate parameters of the caller are passed to the called task, the caller is then temporarily suspended until the called task completes the statements embraced by **do ... end**, and any out parameters are then passed back to the caller. Finally both tasks again procede independently of each other.

We thus see the three basic notions achieved with a rendezvous

- Synchronization. The calling task must issue an entry call, and the called task must reach a corresponding accept statement.

- Exchange of Information. The entry can have parameters, and thus receive or transmit values.

- Mutual Exclusion between two or more processes calling the same entry.

Notice that the rendezvous is named in one direction only. The calling task must know the name of the task containing the entry. The called task on the other hand will accept calls from any task. Thus in general we has a many-to-one pattern of communication. As a consequence, each entry has a queue of tasks calling it. This queue is processed in a strictly first in first out manner and each rendezvous at an accept statement removes just one item from tnis queue.

An accept statement can, of course, be followed by further statements to be executed after a rendezvous.

It shoula be observed that a task can only handle one entry at a time. Although not illustrated by our example, there can be several accept statements for a given entry. We see here a sharp distinction between entries and procedures. All calls of a procedure execute the same body, whereas calls of entries need not.

We now review the complete program for our decoding problem. The program is given in Example 6.1 The main procedure MESSAGE_DECODING starts the action by initiating the three tasks. The three tasks operate quite independently, but are, of course, synchronized through the calls to the entries of DECODE. As given, the three tasks operate forever. Thus the program never terminates.

6.4 Choosing Among Alternative Entry Calls

In our decoding task above, a response to a call of one entry must always be followed by a response to a call of the other entry. In many applications this need not be the case. What if we have two or more entries and want to respond to the first call? What if we cannot predict which call will occur first? More generally, what if we are prepared to accept any one of several entry calls?

For example, let us return to our decoding problem. Observe that once the DECODE task is sent a character code, it must await completion of a call to transmit the decoded value before it can process any more codes. We would really like the generation of codes and the printing of messages to go on much more independently.

Example 6.1 A Solution to the Decoding Problem

```
procedure MESSAGE_DECODING is

   task GENERATE_MESSAGES;

   task DECODE is
      entry SEND_CHAR    (C: in  CHARACTER);
      entry RECEIVE_CHAR(C: out CHARACTER);
   end;

   task PRINT_MESSAGES;


   task body GENERATE_MESSAGES is
      NEXT_CODE: CHARACTER;
   begin
      loop
         -- code for receiving data and
         -- generating NEXT_CODE.

         DECODE.SEND_CHAR(NEXT_CODE);
      end loop;
   end;


   task body DECODE is
      CODE, CHAR: CHARACTER;
   begin
      loop
         accept SEND_CHAR(C: in CHARACTER) do
            CODE := C;
         end;

         -- statements for decoding the value of CODE
         -- and producing the decoded value in CHAR

         accept RECEIVE_CHAR(C: out CHARACTER) do
            C := CHAR;
         end;
      end loop;
   end;
```

```
   task body PRINT_MESSAGES is
      NEXT_CHAR: CHARACTER;
      INDEX    : INTEGER;
      LINE     : STRING(1 .. 120);
   begin
      INDEX := 1;
      loop
         DECODE.RECEIVE_CHAR(NEXT_CHAR);
         LINE(INDEX) := NEXT_CHAR;
         if INDEX < 120 then
            INDEX := INDEX + 1;
         else
            PRINT(LINE);
            INDEX := 1;
         end if;
      end loop;
   end;

begin

   initiate GENERATE_MESSAGES, DECODE, PRINT_MESSAGES;

end MESSAGE_DECODING;
```

In particular, if our printing process is slow, we would still like to accept a burst of entry calls for new input codes. Alternatively, we would like to accept multiple entry calls for printing already received codes while waiting for another (perhaps delayed) input code. For this purpose we can introduce a storage area for characters in the DECODE task. Most importantly for our purposes, as long as the storage area is neither full nor empty, we want to accept a call of either entry.

The basic mechanism for a choice among entry calls is the select statement. Consider the following sketch:

```
   select
      accept SEND_CHAR(C: in CHARACTER) do
         ...
      end;
   or
      accept RECEIVE_CHAR(C: out CHARACTER) do
         ...
      end;
   end select;
```

Such a statement accepts a call to either the entry SEND_CHAR or RECEIVE_CHAR. In particular, when this statement is reached, one of three cases will arise.

(1) Neither entry has been called: in this case the task is suspended until one of the entries is called, in which case the corresponding accept statement is processed.

(2) One (and only one) entry has been called: in this case the corresponding accept statement is immediately processed.

(3) Both entries have been called: in this case one of the entries is accepted. The choice is done at random: either one may be chosen.

As indicated above, we only want to accept a call of SEND_CHAR if there is enough space left in the storage area. Similarly, we only want to accept a call of RECEIVE_CHAR if there are characters in the storage area. These requirements are handled with conditions that guard the alternatives in a select statement. Consider the following sketch:

```
select
   when COUNT < STORAGE_SIZE =>
      accept SEND_CHAR(C: in CHARACTER) do
         ...
      end;

or
   when COUNT > 0 =>
      accept RECEIVE_CHAR(C: out CHARACTER) do
         ...
      end;
end select;
```

The only entries that can be accepted above are those whose guarding conditions evaluate to true.

Finally, we note that as before, an accept statement may be followed by statements to be executed after the rendezvous.

All of the above points are illustrated in Example 6.2, a revised version of the DECODE task given earlier. Note here that the characters are stored in the buffer in a round-robin fashion. The buffer has two indices, an IN_INDEX denoting the space for the next incoming character, and an OUT_INDEX denoting the space for the next character to be transmitted. We also note that the updating of the values of IN_INDEX, OUT_INDEX, and COUNT is not done within the rendezvous. This allows the calling task to continue as soon as possible.

Example 6.2 Putting a Buffer in the Decoding Task

```
task body DECODE is

   STORAGE_SIZE: constant INTEGER := 500;
   COUNT       : INTEGER range 0 .. STORAGE_SIZE;

   IN_INDEX, OUT_INDEX: INTEGER range 1 .. STORAGE_SIZE;

   CODE, CHAR: CHARACTER;

   STORAGE_AREA: array (1 .. STORAGE_SIZE) of CHARACTER;

begin
   COUNT     := 0;
   IN_INDEX  := 1;
   OUT_INDEX := 1;

   loop
         select
            when COUNT < STORAGE_SIZE =>
               accept SEND_CHAR(C: in CHARACTER) do
                  CODE := C;
               end;

               -- statements for decoding the value of CODE
               -- and producing the decoded value in CHAR

               STORAGE_AREA(IN_INDEX) := CHAR;
               COUNT    := COUNT + 1;
               IN_INDEX := (IN_INDEX mod STORAGE_SIZE) + 1;

         or
            when COUNT > 0 =>
               accept RECEIVE_CHAR(C: out CHARACTER);
                  C := BUFFER(OUT_INDEX);
               end;

               COUNT     := COUNT - 1;
               OUT_INDEX := (OUT_INDEX mod STORAGE_SIZE) + 1;
         end select;
      end loop;

end DECODE;
```

6.5 Specifying Delays

The ability to delay a task for a specified time interval is important to many applications. We may want to stop execution for a period of time. Alternatively, we may want to wait for some event to happen, and if the event does not take place within some required time interval, take some different action. Both of these cases are handled by a delay statement.

For example, consider the delay statement

```
delay 10.0*SECONDS;
```

The expression following delay represents the number of basic time units for which the task is to be suspended. This expression is of the predefined floating point type TIME. The predefined constant SECONDS allows such expressions to be written in a natural form. This constant gives the number of basic time units in one second.

A delay statement may occur in place of an accept statement as the synchronization part in a select statement, and may have a guard in the usual way. Such a delay statement is used to provide a time-out for tne select statement. If no rendezvous has occurred within the specified interval, then the statement list following the delay statement is executed. If a rendezvous occurs before the interval has expired, then the delay is cancelled and the select statement is executed normally.

As an example, let us revisit once more our decoding program, in particular the procedure PRINT called by the task PRINT_MESSAGES. This subprogram may in fact call an entry LINE_PRINT for the printing of lines on a chain printer. We obviously do not want to keep the chain going if there is nothing to be printed, especially if the printer may be idle for a considerable period of time. On the other hand, if the printer has been idle, we may need to wait a second for the chain to be fully activated.

Consider then the task PRINTER_DRIVER

```
task PRINTER_DRIVER is
   subtype LINE_IMAGE is STRING(1 .. 120);
   entry LINE_PRINT(L: LINE_IMAGE);
end;
```

and its body

```
task body PRINTER_DRIVER is
   BUFFER      : LINE_IMAGE;
   CHAIN_GOING: BOOLEAN := FALSE;
begin
   loop
      select
         accept LINE_PRINT(L: LINE_IMAGE) do
            BUFFER := L;
         end;

         if not CHAIN_GOING then
            -- start the chain
            delay 1.0*SECONDS;
            CHAIN_GOING := TRUE;
         end if;

      or
         when CHAIN_GOING =>
            delay 10.0*SECONDS;
            -- stop the chain
            CHAIN_GOING := FALSE;
      end select;
   end loop;
end;
```

This task contains the entry LINE_PRINT and controls the operation of the print chain in the manner indicated above. In particular, the if statement

```
if not CHAIN_GOING then
   ...
   delay 1.0*SECONDS;
   ...
end if;
```

specifies the deliberate delay needed to activate the printer.

On the other hand, consider the select statement

```
select
   accept LINE_PRINT(L: LINE_IMAGE) do
      ...
   end;
   ...
or
   when CHAIN_GOING =>
      delay 10.0 SEC;
      ...
end select;
```

As long as there are pending calls to LINE_PRINT, the delay statement is not activated. If no call to LINE_PRINT occurs within 10 seconds, the delay will be activated. Thus the chain will only be stopped if at least ten seconds expire after the last line has been printed.

6.6 Interrupting a Task

On many systems we have hardware interrupts that are triggered by certain events. For example, we may wish to install a STOP button to our decoding system. If no more codes are to be produced, or if for some reason we want our program to terminate, we want to press the STOP button and bring all the tasks to completion.

Hardware interrupts are handled quite simply by interpreting them as external entry calls. A representation specification (see Chapter 8) is used to link the entry with the physical storage address or register in which the interrupt is recorded.

For example, consider the following version of the task body for generating codes:

```
task body GENERATE_MESSAGES is
   NEXT_CODE: CHARACTER;
   entry STOP;
   for STOP use at 8#8060  -- physical address of interrupt
begin
   loop
      select
         accept STOP;
         exit;
      else
         -- code for receiving data and
         -- generating NEXT_CODE

         DECODE.SEND_CHAR(NEXT_CODE);
      end select;
   end loop;

   DECODE.SEND_CHAR(END_OF_TRANSMISSION);
end;
```

Here we see an entry named STOP with no parameters. When a hardware interrupt is recorded and the select statmeent is executed, the statement

```
accept STOP;
```

will treat the interrupt as an entry call. After accepting the STOP signal, the following exit statement will cause termination of the loop. Otherwise, the else part of the select statement will be executed and thus generate another code.

After the loop is terminated, an end of transmission character is sent to the DECODE task, and the GENERATE_MESSAGES task will terminate. The other two tasks can be readily modified so as to terminate when the end of transmission character is sent.

Our final program for the message decoding problem is given in Example 6.3. This completes our tour through tasks, and illustrates most of the features mentioned in this chapter.

Example 6.3 Adding a STOP Button To the Decoding Program

```
procedure MESSAGE_DECODING is

   task GENERATE_MESSAGES;


   task DECODE is
      entry SEND_CHAR    (C: in  CHARACTER);
      entry RECEIVE_CHAR(C: outCHARACTER);
   end;

   task PRINT_MESSAGES


   task body GENERATE_MESSAGES is
      NEXT_CODE: CHARACTER;
      entry STOP;
      for STOP use at 8#8060;
   begin
      loop
         select
            accept STOP;
            exit;
         else
            -- code for receiving data and
            -- generating NEXT_CODE

            DECODE.SEND_CHAR(NEXT_CODE);
         end select;
      end loop;

      DECODE.SEND_CHAR(END_OF_TRANSMISSION);
   end;
```

Example 6.3 (cont)

```
task body DECODE is
   STORAGE_SIZE: constant INTEGER := 500;
   COUNT: INTEGER range 0 .. STORAGE_SIZE;

   IN_INDEX, OUT_INDEX: INTEGER range 1 .. STORAGE_SIZE;

   CODE, CHAR: CHARACTER;

   STORAGE_AREA: array (1 .. STORAGE_SIZE) of CHARACTER;

begin
   COUNT     := 0;
   IN_INDEX  := 1;
   OUT_INDEX := 1;

   loop
      select
         when COUNT < STORAGE_SIZE =>
            accept SEND_CHAR(C: in CHARACTER) do
               CODE := C;
            end;


            -- statements for decoding the value of CODE
            -- and producing the decoded value in char

            STORAGE_AREA(IN_INDEX) := CHAR;
            COUNT    := COUNT + 1;
            IN_INDEX := (IN_INDEX mod STORAGE_SIZE) + 1;

      or
         when COUNT > 0 =>
               accept RECEIVE_CHAR(C: out CHARACTER);
                  C := BUFFER(OUT_INDEX);
               end;
            COUNT     := COUNT - 1;
            OUT_INDEX := (OUT_INDEX mod BUFFER_SIZE) + 1;
         end select;

         exit when CHAR = END_OF_TRANSMISSION and COUNT = 0;
   end loop;

end DECODE;
```

Example 6.2 (cont)

```
task body PRINT_MESSAGES is
   NEXT_CHAR: CHARACTER;
   INDEX    : INTEGER;
   LINE     : STRING(1 .. 120);
begin
   INDEX := 1;
   loop
      DECODE.RECEIVE_CHAR(NEXT_CHAR);
      LINE(INDEX) := NEXT_CHAR;

      if INDEX < 120 then
         INDEX := INDEX + 1;
      else
         PRINT(LINE);
         INDEX := 1;
      end if;

      if NEXT_CHAR = END_OF_TRANSMISSION then
         PRINT(LINE);
         exit;
      end if;
   end loop;
end;


begin -- main program

   initiate GENERATE_MESSAGES, DECODE, PRINT_MESSAGES;

end MESSAGE_DECODING;
```

7. EXCEPTION SITUATIONS

In every application errors may arise. There are many sources of error, and many do not result from an incorrect program. Input data may contain values that are out of range, a hardware unit may fail, a tape may have a parity error, or a transmission line may be sporadically faulty. Normally, a message is reported to the user and the program stops.

The simple termination of a program is not always desirable, and in some cases can be disastrous. A pilot about to land may be using some computer controlled navigational system. Telling him that an overflow error has just caused a shutdown of the entire system is certainly the course of last resort. Such systems often have to do something plausible to keep running.

In this section we discuss the facilities for dealing with such situations. These facilities center on the notion of an exception. An exception is an event that causes suspension of normal program execution. Bringing an exception situation to attention is called raising the exception. Responding to the exception is called handling the exception.

To motivate our discussion, we shall explore the program of Example 7.1. This program computes the inverse for each of 20 matrices. Our interest here centers on the cases where a matrix is singular (if the determinant value is too small), and thus has no inverse.

Example 7.1 **Using Exceptions**

```
procedure INVERT_MATRICES is

   SINGULAR: exception;

   type REAL is digits 10;
   type MATRIX is array (NATURAL, NATURAL) of REAL;

   procedure INVERT(M: in out MATRIX) is
      DETERMINANT: REAL;
      EPSILON    : constant REAL := 1E-10;
   begin
      -- compute the determinant of the matrix;

      if ABS(DETERMINANT) < EPSILON then
         raise SINGULAR;
      end if;

      -- complete computation of the inverse;
   end;


   procedure TREAT_ONE_MATRIX is
      M: MATRIX;
   begin
      READ(M);
      INVERT(M);
      PRINT(M);
   exception
      when SINGULAR => PUT("MATRIX IS SINGULAR");
   end;


begin
   for I in 1 .. 20 loop
      PUT("ITERATION");
      PUT(I);
      TREAT_ONE_MATRIX;
   end loop;
end;
```

7.1 Introducing Exceptions

An exception is introduced by an exception declaration, which gives the name of the exception and defines the scope (i.e. the region of text) in which the exception may be raised. For example, in

```
procedure INVERT_MATRICES is
   SINGULAR: exception;
   ...
end;
```

the exception named SINGULAR is introduced. This exception may be raised during execution of the procedure INVERT_MATRICES. As suggested by the name SINGULAR, the exception situation in mind is the event that some input matrix turns out to be singular.

The language itself defines situations that cause exceptions. These exceptions are the result of errors encountered during program execution. All such runtime errors are treated as predefined exceptions. These include the exceptions

OVERFLOW	when exceeding the maximum implemented value of a number
RANGE_ERROR	when exceeding the declared range of a variable
NO_VALUE_ERROR	when accessing the value of an initialized variable
TASKING_ERROR	when an abnormality occurs during inter-task communication e.g. an attempt to call an entry in an inactive task

7.2 Raising and Handling an Exception

When an error situation occurs during program execution, it must be brought to attention. This is achieved by raising an exception. In our procedure INVERT_MATRICES, execution of the statement

```
raise SINGULAR;
```

causes suspension of normal execution and brings the exception named SINGULAR to attention. The effect is to suspend execution of the program unit or block in which the exception arises.

The response to an exception will, of course, vary according to the application. The response can range from doing nothing, in which case the program will usually be terminated, to taking elaborate steps to deal with the situation causing the exception before resuming program execution.

A response to an exception is achieved by appending one or more exception handlers to the end of a block or program unit. When an exception is raised in the unit and normal execution is suspended, the corresponding exception handler is executed. Execution of the handler thus replaces execution of the remainder of the unit.

A block or program unit giving a response to one or more exceptions always ends with an exception part of the form

```
exception
   -- sequence of exception handlers
end;
```

For example, we may have

```
exception
   when SINGULAR => PUT("MATRIX IS SINGULAR");
end;
```

or

```
exception

   when OVERFLOW =>
      X := X_MAX;
      Y := Y_MAX;
      Z := Z_MAX;

   when SENSOR_ERROR =>
      X := X_AVE;
      Y := Y_AVE;
      Z := Z_AVE;

end;
```

We see that an exception handler has a form similar to an alternative in a case statement,

```
when exception_name => statements
```

Consider the simple procedure

```
procedure GET_NEXT_POINT(X,Y,Z: out COORDINATE) is
   -- local declarations
begin

   -- code to read sensor values and obtain the relative
   -- X, Y, and Z coordinates of a point.
   -- The exceptions OVERFLOW and SENSOR_ERROR may be raised
   -- during execution.

exception
   when OVERFLOW =>
      X := X_MAX;
      Y := Y_MAX;
      Z := Z_MAX;

   when SENSOR_ERROR =>
      X := X_AVE;
      Y := Y_AVE;
      Z :- Z_AVE;
end;
```

Here we have a procedure whose execution can result in the raising of an exception and an attempt at some repair to the situation. When the exception OVERFLOW is raised, the parameters X, Y, and Z are assigned maximum values; when the exception SENSOR_ERROR is raised, the parameters are assigned average values. In both cases, after execution of the handler program execution is resumed at the point of call to the procedure.

7.3 Propagation of an Exception

In general, an exception can be raised in a block or program unit and an exception handler can be used to complete the execution of the unit. Frequently however, a block or program unit will have no handler for an exception. Even when exception handlers are provided, they may not cover every exception that might be raised during execution. Furthermore, an exception handler itself may raise an exception. In each of these cases, the block or program unit is terminated, and the exception is said to be <u>propagated</u>.

For example, consider the procedure

```
procedure INVERT(M: in out MATRIX) is
   DETERMINANT: REAL;
   EPSILON    : constant REAL := 1E-10;
begin
   -- compute the determinant of the matrix;

   if ABS(DETERMINANT) < EPSILON then
      raise SINGULAR;
   end if;

   -- complete computation of the inverse;
end;
```

and the calling sequence given in TREAT_ONE_MATRIX

```
READ(M);
INVERT(M);  -- point of call to INVERT
PRINT(M);
```

The computations of INVERT may result in a too small determinant, in which case the exception SINGULAR is raised. As no handler is provided in INVERT, the execution of INVERT is terminated and the exception SINGULAR is (implicitly) raised at the statement where INVERT is called. This implicit raising of an exception is called an exception propagation.

The predefined language exceptions can be viewed as exceptions propagated by the built-in features of the language. For example, when computing

```
X*Y
```

if the result exceeds the maximum implemented value, the exception OVERFLOW is propagated by the operation * at the point where the multiplication is written. When an exception is propagated, the calling unit may, of course, provide a handler for the exception. For example, consider the procedure TREAT_ONE_MATRIX.

```
procedure TREAT_ONE_MATRIX is
   M: MATRIX;
begin
   READ(M);
   INVERT(M);
   PRINT(M);
exception
   when SINGULAR => PUT("MATRIX IS SINGULAR");
end;
```

Here, if a call to INVERT results in the propagation of the exception SINGULAR, the exception is handled in the procedure TREAT_ONE_MATRIX. The handler prints a message in response to the exception. This completes the execution of the procedure

TREAT_ONE_MATRIX, and program execution continues at the point where TREAT_ONE_MATRIX itself is called.

Observe that if any other exception is propagated from INVERT, for example RANGE_ERROR, the exception is again propagated, this time to the point of call for TREAT_ONE_MATRIX.

We thus see here the general rule regarding the propagation of exceptions: a raised exception is propagated through the sequence of invoking units until a unit provides a handler for the exception.

In the simple case where a program provides no handler for an exception and an execution error results in raising the exception, the program is ultimately terminated. Normally, before termination the predefined environment for an implementation will provide some response to every exception, usually the printing of a diagnostic. This is the familiar run time error message.

We now review our program to invert matrices, as given in Example 7.1. This program successivly reads in the values for one of the 20 matrices, inverts the matrix, and prints its inverse.

The computations of a too small determinant will raise the exception SINGULAR, in which case the procedure INVERT is terminated and the exception is propagated to the calling procedure TREAT_ONE_MATRIX. This procedure in turn is suspended, but here a handler for the exception is provided. The handler simply prints the message

```
MATRIX IS SINGULAR
```

This action terminates execution of TREAT_ONE_MATRIX, and then control is returned to its point of call, where the next matrix is processed. Thus we see here the propagation of an exception and a simple diagnostic action, allowing the program to continue in case of an error.

Finally, we note that the raising of any other exception within INVERT, for example RANGE_ERROR, is propagated back to each caller. In this case the program is terminated; presumably, the default environment for an implementation will give some diagnostic message.

It is possible to make our program more robust by providing a default handler for all possible other errors known or unknown

```
procedure TREAT_ONE_MATRIX is
   M: MATRIX;
begin
   READ(M);
   INVERT(M);
   PRINT(M);
exception
```

```
        when SINGULAR => PUT("MATRIX IS SINGULAR");
        when others => PUT("UNEXPECTED ERROR");
    end;
```

The reserved word **others** here stands for every other exception name. Here it will catch every exception except SINGULAR. As a consequence it will be possible to process the next matrix, for example in use of OVERFLOW or RANGE_ERROR, etc.

7.4 Exceptions Arising During Inter-task Communication

The notion of multiple tasks brings up the issue of communicating program units. Of interest here is that an exception situation in one task may be relevant to another task. Faulty communication between tasks is generally brought to attention with the predefined exception TASKING_ERROR.

For example, consider the following cases

(a) a task calls an entry in another task, but the second task has already terminated

(b) a task calls an entry in another task, the second task is active but terminates before accepting the entry call.

(c) a task calls an entry in another task, the second task accepts the entry call but an unhandled exception arises during the rendezvous.

In the first two cases, the exception TASKING_ERROR is raised in the calling task. In the third case, the unhandled exception is also raised in the calling task.

Finally, a task may discover that another task is faulty and that communication by normal means (that is, by entry calls) has become impossible. In such an extreme and abnormal situation, the predefined exception FAILURE can be explicitly raised for the second task. Just as for any exception, a handler for FAILURE in the second task can take appropriate action.

8. UTILIZING THE IMPLEMENTATION

A program does not exist alone. It must be translated into a form suitable for machine execution, and executed on a specific machine. In most cases, a high level language permits the formulation of the program in terms that do not depend on the machine. The main benefit derived from this machine independence is portability. However there are situations in which it is important to have some control over the implementation and to take advantage of its characteristics.

8.1 Representing Data

In Chapter 2 we discussed the basic notion of types, whereby the logical properties of data are defined. Some system applications require the ability to describe the physical layout of data, either for dealing with special hardware devices or for efficiency. We turn then to the notion of representation specifications.

The facility for representation specifications is based on two underlying ideas:

- Separation Principle: Data can be specified in two steps. First, the logical properties are described via a type definition. Second, any special representation properties are described by a representation specification for the type.

- One Type, One Representation: A given type of data can have only one explicit representation.

In the usual case, the programmer is not concerned with the specific representation of data, and the representation is chosen by the translator.

Consider the following type declaration and associated representation specification:

```
type BIT_MAP is array (1 .. 100,  1 .. 100) of BOOLEAN;

for  BIT_MAP use packing;
```

In the absence of a representation specification, no assumption can be made on the representation chosen by the translator. Here, however, the array type BIT_MAP is explicitly defined as having a packed representation. In this case, all objects declared with the type BIT_MAP will be represented as packed arrays.

More generally, a representation specification can be used to specify detailed bit configurations for data. For example, consider the following declarations:

```
WORD: constant INTEGER := 4;  -- storage unit is byte, 4 bytes per word

type STATE is (A, M, W, P);
type MODE  is (FIX, DEC, EXP, SIGNIF);

type PROGRAM_STATUS_WORD is
   record
      SYSTEM_MASK     : array(0 .. 7) of BOOLEAN;
      PROTECTION_KEY  : INTEGER range 0 .. 3;
      MACHINE_STATE   : array(STATE'FIRST .. STATE'LAST) of BOOLEAN;
      INTERRUPT_CAUSE: INTERRUPTION_CODE;
      ILC             : INTEGER range 0 .. 3;
      CC              : INTEGER range 0 .. 3;
      PROGRAM_MASK    : array(MODE'FIRST .. MODE'LAST) of BOOLEAN;
      INST_ADDRESS    : ADDRESS;
   end record;
```

Objects of type PROGRAM_STATUS_WORD can be given a specific representation as follows:

```
for PROGRAM_STATUS_WORD use
   record
      SYSTEM_MASK     at 0*WORD range 0  .. 7;
      PROTECTION_KEY  at 0*WORD range 10 .. 11;  -- bits 8, 9 unused
      MACHINE_STATE   at 0*WORD range 12 .. 15;
      INTERRUPT_CAUSE at 0*WORD range 16 .. 31;
      ILC             at 1*WORD range 0  .. 1;   -- second word
      CC              at 1*WORD range 2  .. 3;
      PROGRAM_MASK    at 1*WORD range 4  .. 7;
      INST_ADDRESS    at 1*WORD range 8  .. 31;
   end record;
```

In each of the above cases, we see the general form for specifying the representation of data

for *type name* **use** *representation*;

For the first example above, the representation is specified simply by the keyword **packing**. In the second example, the specific bit configuration of each record component is specified.

For records, each component representation has the form

component name **at address range** *first bit* .. *last bit*;

The component address is the relative address (in storage units) with respect to the origin of the record object; the first bit and the last bit define the bit positions of the component within the storage unit. In the case above, for each object of type PROGRAM_STATUS_WORD, the first eight bits (bits 0 through 7 of word 0) are occupied by the component SYSTEM_MASK, bits 10 and 11 by the component PROTECTION_KEY, etc.

There are several other forms of representation specifications in Green. For instance we can specify the codes for representing the values of an enumeration type or the storage address of a variable. Each form illustrates the same general idea: the programmer can exercise some control over the characteristics of the machine on which a program is executed.

8.2 Changing the Representation of Data

A classical problem in dealing with data is that we may want to change its representation. For example, data stored on a given device may need to be moved to a different device, or we may wish to change data stored in a compact form into another form for more efficient processing.

The problem of change of representation is straightforward; it can be expressed as an explicit type conversion between a first type and a second type derived from the first but with different representation specifications.

For example, consider

```
type DESCRIPTOR is
   record
      -- components of a descriptor
   end;

type PACKED_DESCRIPTOR is new DESCRIPTOR;

for PACKED_DESCRIPTOR use packing;
```

DESCRIPTOR and PACKED_DESCRIPTOR are two different types with identical characteristics, apart from their representation. Change of representation can be accomplished with explicit type conversions. Thus with the declarations:

```
D: DESCRIPTOR;
P: PACKED_DESCRIPTOR;
```

we can write

```
P := PACKED_DESCRIPTOR(D);   -- pack
D := DESCRIPTOR(P);          -- unpack
```

Here the type conversions are specified by expressions of the form

derived type(value)

Such a conversion accomplishes a change of representation of the value to that given for the derived type.

8.3 Giving Instructions to the Translator

The creation of a program involves some communication with the language translator. For example, we may wish to state that only certain portions of a program unit are to be listed, or we may wish to specify that the code generated for a subprogram is to be inserted in line. This gives rise to the notion of a pragma.

A pragma is an instruction to the translator. Since there may be many translators for a language, the allowable pragmas will generally vary from implementation to implementation. Some pragmas however, for example the pragma specifying that a subprogram is to be expanded in line, are predefined for all implementations. The following examples illustrate use of pragmas that are predefined in the language:

```
pragma LIST(OFF);                    -- suspend listing

pragma INCLUDE("PROJECT_INFO");      -- include the text file PROJECT_INFO
                                     -- where the pragma is given

pragma MEMORY_SIZE(60_000);          -- the available memory is limited
                                     -- to 60,000 storage units

pragma INLINE;                       -- at each call, expand the subprogram
                                     -- in which the pragma appears
```

A typical implementation may also have predefined pragmas, for example

```
pragma TRACE(X, Y);          -- generate code to monitor changes
                             -- to the variables X and Y

pragma PROFILE(STORAGE);     -- provide a storage profile for
                             -- the program
```

Again, in each case, the point is the same. a pragma allows the programmer to exercise some control over the characteristics of the translator that processes a program.

8.4 Environment Inquiries

In Chapter 2 on describing data we discussed the notion of the predefined attributes of a type. More generally, the predefined attributes serve as a mechanism for obtaining information that is known to the translator.

All predefined attributes have the form

name'attribute identifier

The name is that of some program entity, the identifier is one of the predefined attribute identifiers for the entity.

There are several broad uses of predefined attributes. One use is to refer to information about types, for example:

```
DAY'FIRST            -- the first element of type DAY
DAY'SUCC             -- the successor function for the type DAY

COEFFICIENT'DIGITS   -- the number of decimal DIGITS specified in the
                     -- declaration of the type COEFFICIENT

COEFFICIENT'LARGE    -- the maximum expressible value of
                     -- type COEFFICIENT

INTEGER'SIZE         -- the implemented size of integers in bits
```

A second class of uses is for referring to general information known to the translator. For example if PSW is a variable of type PROGRAM_STATUS_WORD, we may have

```
PROGRAM_STATUS_WORD'SIZE    -- the number of bits specified for
                            -- representing objects of type
                            -- PROGRAM_STATUS_WORD

PSW'ADDRESS                 -- the storage address of the variable PSW

PSW.PROGRAM_MASK'FIRST_BIT  -- the first bit of the record
                            -- component PSW.PROGRAM_MASK
```

A third class of use is for obtaining information that is known during program execution. For example, we may have

```
DECODE'ACTIVE  -- true if the task DECODE has been
               -- initiated but has not yet terminated

DECODE'TIME    -- the cumulative processing time
               -- since the task DECODE has been initiated.
```

In addition to language predefined attributes, an implementation may introduce other implementation specific predefined attributes. Overall, the facility for predefined attributes provides a method for obtaining program and implementation dependent information.

9. LARGE PROGRAMS

As programs become larger, the problems in dealing with them become much larger. As a result, problems such as

- the organization of pages and pages of program text
- the coordination of the work of several authors
- the control of errors
- the costs and difficulties of making modifications

are all too familiar.

We treat here the notion of separate compilation, a major facility for controlling program development. While useful even for small programs, for large programs the notion of separate compilation is central. Its basic goal is to permit the division of large programs into simpler, more manageable parts.

For illustration, we consider the development of a text editing system for producing files of text. Such an application may involve implementing numerous commands for entering and changing text, facilities for obtaining and deleting files, as well as general requests for obtaining informationn or even printing the time of day. Typically such applications require many thousands of lines of code. In short, they are large and complex.

9.1 The Notion of Separate Compilation

Consider for the moment the development of a single compilation unit for such a large scale program, as sketched very briefly in Example 9.1. This program sketch illustrates the use of several package modules and numerous procedures. We also see the textual nesting of procedures and the use of visibility restrictions in attempting to control the effect of program units.

Example 9.1 An Editor as a Single Compilation Unit

```
procedure EDITOR is

   package COMMANDS is
      MAX_KEY_LENGTH: constant INTEGER := 10;
      REQUEST_PREFIX: constant STRING  := "--";
      type PATTERN    is ...;
      type KEY_SYMBOL is ...;
      type REQUEST is
         record
            NAME         : KEY_SYMBOL;
            REPEATER     : INTEGER;
            TEXT_STR     : PATTERN;
            NEW_TEXT_STR: PATTERN;
         end record;
   end;

   package STATE_DATA is
      type LINE_PTR is new INTEGER;
      type STATUS is ...;
      CURRENT_LINE: LINE_PTR;
      ASSUMPTIONS : array (STATUS'FIRST .. STATUS'LAST) of BOOLEAN;
   end;

   package FILE_DATA is
      NUM_FILES: constant INTEGER := 100;
      type FILE_STATUS is (OLDF, NEWF);
      type FILE_DESCRIPTOR is ...;
      FILE_DIRECTORY: array (1 .. NUM_FILES) of FILE_DESCRIPTOR;
   end;

   procedure GET_REQUEST(R: out COMMANDS.REQUEST) is
      use COMMANDS;
      -- local declarations;

      type LINE_IMAGE is STRING(1 .. 120);

      procedure GET_LINE(LINE       : out LINE_IMAGE;
                         LINE_LENGTH: out INTEGER);

      procedure TOKENIZE(LINE       : in  LINE_IMAGE;
                         LINE_LENGTH: in  INTEGER;
                         REQUEST    : out REQUEST);

      -- bodies of procedures

   begin
      -- obtain and parse the next request
   end;
```

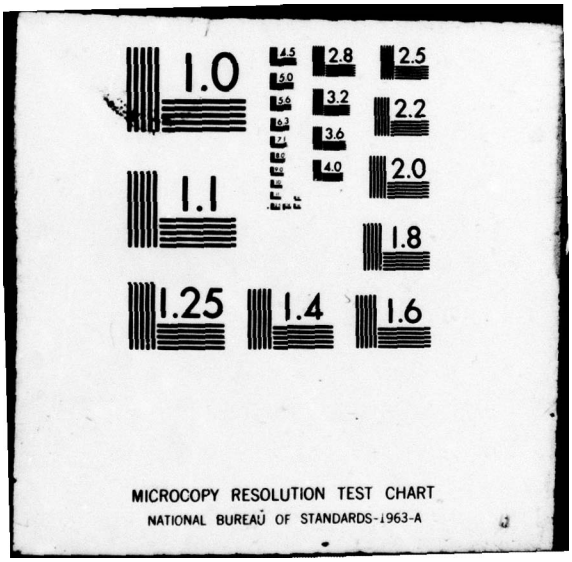
```
   procedure INITIALIZE_USER is
   use FILE_DATA, STATE_DATA;
      -- set up initial user environment
   end;


   procedure PERFORM_REQUEST(R: in COMMANDS.REQUEST) is
      use COMMANDS, STATE_DATA;
      -- local declarations

      subtype LINE_PTR is INTEGER range 0 .. 120;

      procedure PERFORM_LIST   (NUM_LINES      : in  INTEGER;
                                SEARCH_STR     : in  PATTERN;
                                START_POSITION: in  LINE_PTR);

      procedure PERFORM_DELETE(NUM_LINES      : in  INTEGER;
                               SEARCH_STR     : in  PATTERN;
                               START_POSITION: in  LINE_PTR;
                               END_POSITION   : out LINE_PTR;
      ...

   end;

   restricted (EDITOR, TERMINAL_I_O_PACKAGE)
   procedure DISPLAY_ERROR(ERROR_CODE: INTEGER);
      ...
   end;


   restricted (EDITOR, TERMINAL_I_O_PACKAGE)
   procedure PRINT_DATE is
      ...
   end;

   ...


begin  -- main program

   -- code for setting up and executing the editor

end EDITOR;
```

AD-A070 753 HONEYWELL INC MINNEAPOLIS MINN SYSTEMS AND RESEARCH --ETC F/G 9/2
THE GREEN LANGUAGE: AN INFORMAL INTRODUCTION.(U)
APR 79 MDA903-77-C-0331

UNCLASSIFIED NL

2 OF 2
AD
A070753

END
DATE
FILMED
8-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

In our discussion of program units in Chapter 4, we noted that all program units can be specified in two parts

(a) A declaration, summarizing the calling conventions of a unit, and

(b) A body, specifying the implementation.

The body of a program unit can be compiled as a separate compilation unit. This applies to both subprogram bodies and module bodies. In addition, module specifications can also be separately compiled.

We also recall that program units can be restricted, and that the use of any information in other program units can be stated in a visibility restriction prefixing the unit. These dependencies may be through shared modules, or through the textual nesting of subprograms. This gives rise to a general rule for separate compilation.

- Every compilation unit is in effect a restricted program unit. In the absence of an explicit visibility restriction, an empty visibility list is assumed.

In particular, any compilation unit that utilizes the information in other compilation units must name each unit in a visibility restriction.

Separate compilation is illustrated in Example 9.2, which gives a sketch of the previously given editor example but with extensive use of the facility for separate compilation. Note how visibility lists have been included for some units to make explicit their dependence on other units.

The broken lines between compilation units are here to remind the reader that these units need not be contiguous texts. The compilation units outlined here collectively define a single programming system, and are hence said to form a program library. We now discuss in more detail the notions of a program library and separate compilation.

Example 9.2 An Editor with Multiple Compilation Units

```
package COMMANDS is
   MAX_KEY_LENGTH: constant INTEGER := 10;
   REQUEST_PREFIX: constant STRING  := "--";
   type PATTERN     is ...;
   type KEY_SYMBOL is ...;
   type REQUEST is
      record
         NAME         : KEY_SYMBOL;
         REPEATER     : INTEGER;
         TEXT_STR     : PATTERN;
         NEW_TEXT_STR: PATTERN;
      end record;
end;

-------------------------------------------------

package STATE_DATA is
   type LINE_PTR is new INTEGER;
   type STATUS is ...;
   CURRENT_LINE: LINE_PTR;
   ASSUMPTIONS : array (STATUS'FIRST .. STATUS'LAST) of BOOLEAN;
end;

-------------------------------------------------

package FILE_DATA is
   NUM_FILES: constant INTEGER := 100;
   type FILE_STATUS is (OLDF, NEWF)
   type FILE_DESCRIPTOR is ...;
   FILE_DIRECTORY: array (1 .. NUM_FILES) of FILE_DESCRIPTOR
end;

-------------------------------------------------

restricted (COMMANDS)
procedure EDITOR is
   -- local declarations

   procedure INITIALIZE_USER is separate;
   procedure GET_REQUEST    (R: out COMMANDS.REQUEST) is separate;
   procedure PERFORM_REQUEST(R: in  COMMANDS.REQUEST) is separate;
   procedure DISPLAY_ERROR(ERROR_CODE: INTEGER) is separate;
   procedure PRINT_DATE is separate;

begin
   -- code for setting up and executing the editing system
end;

-------------------------------------------------
```

```
restricted (EDITOR, FILE_DATA, STATE_DATA)
separate procedure INITIALIZE_USER is
   use FILE_DATA, STATE_DATA;
   -- set up initial user environment
end;

-------------------------------------------------

restricted (EDITOR, COMMANDS)
separate procedure GET_REQUEST(R : out COMMANDS.REQUEST) is
   use COMMANDS;
   -- local declarations
   type LINE_IMAGE is STRING(1 .. 120);

   procedure GET_LINE(LINE       : out LINE_IMAGE;
                      LINE_LENGTH: out INTEGER) is separate;

   procedure TOKENIZE(LINE       : in  LINE_IMAGE;
                      LINE_LENGTH: in INTEGER;
                      REQUEST    : out REQUEST) is separate;
   ...
begin
   -- obtain and parse the next request
end;

-------------------------------------------------

restricted (EDITOR, COMMANDS, STATE_DATA)
separate procedure PERFORM_REQUEST(R: in COMMANDS.REQUEST) is
   use COMMANDS, STATE_DATA;
   subtype LINE_PTR is INTEGER range 0 .. 120;
   -- local declarations

   procedure PERFORM_LIST(NUM_LINES     : in INTEGER;
                          SEARCH_STR    : in PATTERN;
                          START_POSITION: in LINE_PTR) is separate;

   procedure PERFORM_DELETE(NUM_LINES     : in  INTEGER;
                            SEARCH_STR    : in  PATTERN;
                            START_POSITION: in  LINE_PTR;
                            END_POSITION  : out LINE_PTR) is separate;
   ...
begin
   -- code for executing a given request
end;

-------------------------------------------------

restricted (EDITOR, TERMINAL_I_O_PACKAGE)
procedure DISPLAY_ERROR(ERROR_CODE: INTEGER) is
   ...
end;
```

```
-------------------------------------------------

restricted(EDITOR, TERMINAL_I_O_PACKAGE)
procedure PRINT_DATE is
   ...
end;

-------------------------------------------------

   ...
```

9.2 Translation of Generally Usable Packages

In any large program, we want to define packages to be used by several units. As discussed earlier, packages of related data, types, and subprograms can be created. Such packages can be separately compiled. The name of a separately compiled package can be made visible to another compilation unit in the program library by including the package name in the visibility list of the unit.

For example in our editor sketch, we may separately compile

```
package COMMANDS is
   ...
end;
```

This package module is a stand alone collection of data and types, and thus has no visibility clause. On the other hand, the module can be made accessible to other units, for example in the main procedure

```
restricted (COMMANDS)
procedure EDITOR is
   ...
end;
```

Packages will be present in the program library when they have been compiled there (as in the case of COMMANDS). The programming environment will also include commands for transferring generally usable packages from one program library to another. For example, this ability might be used for a general package containing input-output routines for terminals.

```
package TERMINAL_I_O_PACKAGE is
   procedure INHIBIT_ERROR_EXITS;
   procedure DISABLE_INTERRUPTS;
   ...
end;

package body TERMINAL_I_O_PACKAGE is
   -- implementation of system routines
end;
```

Any program unit utilizing these procedures must also name the module in its visibility clause, for example,

```
restricted (EDITOR, TERMINAL_I_O_PACKAGE)
separate procedure DISPLAY_ERROR(ERROR_CODE: INTEGER) is
   ...
end;
```

Both of these cases illustrate a general point. Packages of common information can be developed separately, and then selectively made available to other separately developed compilation units. This corresponds to bottom-up program development.

9.3 Top Down Program Development

Programs can be developed and organized in various ways. One way is to write the program top down, that is write the upper level subprograms first, then write the subprograms needed to define the main subprograms, etc. The facility for separate compilation supports this approach.

For example consider the upper level procedure of our editor:

```
restricted (COMMANDS)
procedure EDITOR is
   -- local declarations

   procedure INITIALIZE_USER is separate
   procedure GET_REQUEST    (R: out COMMANDS.REQUEST) is separate;
   procedure PERFORM_REQUEST(R: in  COMMANDS.REQUEST) is separate;
   procedure DISPLAY_ERROR(ERROR_CODE: INTEGER) is separate;
   procedure PRINT_DATE is separate;

begin
   -- code for setting up and executing the editing system
end;
```

This procedure can be written in its entirety, except for the bodies of the five internal procedures. These five procedure bodies can be developed later, as separate texts. Such a separation is marked by providing body stubs marked by the reserved word separate as in

```
procedure GET_REQUEST(REQUEST: out REQUEST) is separate;
```

When the body is later completed, it can be submitted to the translator as a compilation unit for the given program library.

Just as for procedures that are not separately compiled, a body can make use of information in outer program units. The same possibility holds for compilation units. In such a case, the visibility clause for the separate procedure body must name the compilation unit in which the procedure is declared.

For example, the body of the procedure GET_REQUEST can be separately compiled as

```
restricted (EDITOR, COMMANDS)
separate procedure GET_REQUEST(R: COMMANDS.REQUEST) is
   ...
end;
```

Here the body of GET_REQUEST utilizes the declarative information of the main procedure EDITOR as well as the package module COMMANDS.

Although not illustrated by our editor example, this same facility applies to module bodies. In particular, the visible part of a module can be defined within a compilation unit and its module body can be specified as separate. The body can be later written and compiled separately.

In summary, we see that the notion of separate compilation supports several important techniques of program development, including the use shared information and libraries, top down development, and bottom up development. In all cases, once the interface for a compilation unit is spelled out, the unit can be developed independently of the other compilation units.

9.4 Order of Compilation

We need to submit the compilation units of a program library in an order that is understandable to a translator. Otherwise, we would like as much freedom as possible.

The rules for the order in which compilation units must be compiled follow directly from the visibility rules.

- A compilation unit may only be compiled after all units that are visible in the compilation unit have been compiled.

Otherwise, the order of compilation is arbitrary, and can be left to the programmer.

The effect of this simple rule can be illustrated with the program sketch of Example 9.2. For example:

(1) The package module COMMANDS must be compiled before the main procedure EDITOR.

(2) The package modules STATE_DATA and FILE_DATA must be compiled before the procedure INITIALIZE_USER.

(3) The package module COMMANDS and the main procedure EDITOR must be compiled before the procedure GET_REQUEST.

Apart from required ordering on compilation, there is considerable choice remaining for the programmer. For example:

(1) The three package modules COMMANDS, STATE_DATA, and FILE_DATA can be compiled before or after each other

(2) The three procedures INITIALIZE_USER, GET_REQUEST, and PERFORM_REQUEST can be compiled before or after each other.

In any programming system, correction or program updates require recompilation of compilation units. In large systems, there may even be thousands of modifications to a program. Obviously, in retranslating a unit we may change some information that is visible to other units, and hence any unit that utilizes this information can be affected by the change.

The rule for recompilation also follows the visibility rules:

- a compilaion unit needs to be recompiled whenever a unit that is visible in the compilation unit is recompiled.

Otherwise, no further recompilations are required.

In the sketch of our editing program for instance, recompilation of the definition module FILE_DATA requires recompilation of the procedure INITIALIZE_USER. Recompilation of the main procedure EDITOR requires recompilation of the procedures GET_REQUEST and PERFORM_REQUEST. Otherwise, no other units in our sketch need to be recompiled.

We see here that with a suitable organization of compilation units, the effect of program changes can be tightly controlled. Moreover, a change to a given unit need not necessitate a cascade of recompilations of other units.